Zephyr Project Documentation

Release 2.7.5

The Zephyr Project Contributors

Jun 01, 2023

Table of contents

Chapter 1

Introduction

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

The Zephyr kernel supports multiple architectures, including:

- ARC EM and HS
- ARMv6-M, ARMv7-M, and ARMv8-M (Cortex-M)
- ARMv7-A and ARMv8-A (Cortex-A, 32- and 64-bit)
- ARMv7-R, ARMv8-R (Cortex-R, 32- and 64-bit)
- Intel x86 (32- and 64-bit)
- NIOS II Gen 2
- RISC-V (32- and 64-bit)
- SPARC V8
- Tensilica Xtensa

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

1.1 Licensing

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

1.2 Distinguishing Features

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

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

- *Multi-threading Services* for cooperative, priority-based, non-preemptive, and preemptive threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.
- *Interrupt Services* for compile-time registration of interrupt handlers.
- *Memory Allocation Services* for dynamic allocation and freeing of fixed-size or variable-size memory blocks.

- *Inter-thread Synchronization Services* for binary semaphores, counting semaphores, and mutex semaphores.
- *Inter-thread Data Passing Services* for basic message queues, enhanced message queues, and byte streams.
- *Power Management Services* such as tickless idle and an advanced idling infrastructure.

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

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

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

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

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

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

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

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

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

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

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

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

- Verified with multiple popular controllers
- Highly configurable

Mesh Support:

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

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

Virtual File System Interface with LittleFS and FATFS Support LittleFS and FATFS Support, FCB (Flash Circular Buffer) for memory constrained applications, and file system enhancements for logging and configuration.

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

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

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

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

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

1.3 Community Support

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

1.4 Resources

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

- **Help**: Asking for Help Tips
- **Documentation**: http://docs.zephyrproject.org (Getting Started Guide)
- **Source Code**: https://github.com/zephyrproject-rtos/zephyr is the main repository; https://elixir.bootlin.com/zephyr/latest/source contains a searchable index
- **Releases**: https://github.com/zephyrproject-rtos/zephyr/releases
- **Samples and example code**: see Sample and Demo Code Examples
- **Mailing Lists**: users@lists.zephyrproject.org and devel@lists.zephyrproject.org are the main user and developer mailing lists, respectively. You can join the developer's list and search its archives at Zephyr Development mailing list. The other Zephyr mailing list subgroups have their own archives and sign-up pages.
- **Nightly CI Build Status**: https://lists.zephyrproject.org/g/builds The builds@lists.zephyrproject.org mailing list archives the CI nightly build results.

- **Chat**: Real-time chat happens in Zephyr's Discord Server. Use this Discord Invite to register.
- **Contributing**: see the Contribution Guide
- **Wiki**: Zephyr GitHub wiki
- **Issues**: https://github.com/zephyrproject-rtos/zephyr/issues
- **Security Issues**: Email vulnerabilities@zephyrproject.org to report security issues; also see our Security documentation. Security issues are tracked separately at https://zephyrprojectsec.atlassian.net.
- **Zephyr Project Website**: https://zephyrproject.org

1.5 Fundamental Terms and Concepts

See glossary

Chapter 2

Getting Started Guide

Follow this guide to:

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

2.1 Select and Update OS

Click the operating system you are using.

Ubuntu

This guide covers Ubuntu version 18.04 LTS and later.

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

macOS

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

On other versions, see this Apple support topic.

Windows

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

2.2 Install dependencies

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

The current minimum required version for the main dependencies are:

Tool	Min. Version
CMake	3.20.0
Python	3.6
Devicetree compiler	1.4.6

Ubuntu

1. Download, inspect and execute the Kitware archive script to add the Kitware APT repository to your sources list. A detailed explanation of `kitware-archive.sh` can be found here kitware third-party apt repository:

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

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

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

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

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

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

macOS

1. Install Homebrew:

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

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

```
brew install cmake ninja gperf python3 ccache qemu dtc
```

Windows

Note: Due to issues finding executables, the Zephyr Project doesn't currently support application flashing using the Windows Subsystem for Linux (WSL) (WSL).

Therefore, we don't recommend using WSL when getting started.

These instructions must be run in a `cmd.exe` command prompt. The required commands differ on PowerShell.

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

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

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

4. Use `choco` to install the required dependencies:

```
choco install cmake --installargs 'ADD_CMAKE_TO_PATH=System'
choco install ninja gperf python git dtc-msys2
```

5. Close the window and open a new `cmd.exe` window **as a regular user** to continue.

2.3 Get Zephyr and install Python dependencies

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

Python is used by the `west` meta-tool as well as by many scripts invoked by the build system. It is easy to run into package incompatibilities when installing dependencies at a system or user level. This situation can happen, for example, if working on multiple Zephyr versions at the same time. For this reason it is suggested to use Python virtual environments.

Ubuntu

Install globally

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

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

2. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

Install within virtual environment

1. Create a new virtual environment:

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

2. Activate the virtual environment:

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

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

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

---

3. Install west:

```
pip install west
```

4. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

macOS

Install globally

1. Install west:

```
pip3 install -U west
```

2. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

Install within virtual environment

1. Create a new virtual environment:

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

2. Activate the virtual environment:

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

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

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

---

3. Install west:

```
pip install west
```

4. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

Windows

Install globally

1. Install west:

```
pip3 install -U west
```

2. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

Install within virtual environment

1. Create a new virtual environment:

```
cd %HOMEPATH%
python3 -m venv zephyrproject\.venv
```

2. Activate the virtual environment:

```
:: cmd.exe
zephyrproject\.venv\Scripts\activate.bat
:: PowerShell
zephyrproject\.venv\Scripts\Activate.ps1
```

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

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

---

3. Install west:

```
pip install west
```

4. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

2.4 Install a Toolchain

A toolchain provides a compiler, assembler, linker, and other programs required to build Zephyr applications.

Ubuntu

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

1. Download the latest SDK installer:

```
cd ~
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.13.1/
↪zephyr-sdk-0.13.1-linux-x86_64-setup.run
```

2. Run the installer, installing the SDK in `~/zephyr-sdk-0.13.1`:

```
chmod +x zephyr-sdk-0.13.1-linux-x86_64-setup.run
./zephyr-sdk-0.13.1-linux-x86_64-setup.run -- -d ~/zephyr-sdk-0.13.1
```

Note: It is recommended to install the Zephyr SDK at one of the following locations:

- `$HOME/zephyr-sdk[-x.y.z]`
- `$HOME/.local/zephyr-sdk[-x.y.z]`
- `$HOME/.local/opt/zephyr-sdk[-x.y.z]`
- `$HOME/bin/zephyr-sdk[-x.y.z]`
- `/opt/zephyr-sdk[-x.y.z]`
- `/usr/zephyr-sdk[-x.y.z]`
- `/usr/local/zephyr-sdk[-x.y.z]`

where `[-x.y.z]` is optional text, and can be any text, for example `-0.13.1`.

If installing the Zephyr SDK outside any of those locations, please read: *Install the Zephyr Software Development Kit (SDK)*

You cannot move the SDK directory after you have installed it.

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

```
sudo cp ~/zephyr-sdk-0.13.1/sysroots/x86_64-pokysdk-linux/usr/share/openocd/
↪contrib/60-openocd.rules /etc/udev/rules.d
sudo udevadm control --reload
```

macOS

Follow the instructions in *Set Up a Toolchain*. Note that the Zephyr SDK is not available on macOS.

Do not forget to set the required *environment variables* (`ZEPHYR_TOOLCHAIN_VARIANT` and toolchain specific ones).

Windows

Follow the instructions in *Set Up a Toolchain*. Note that the Zephyr SDK is not available on Windows.

Do not forget to set the required *environment variables* (`ZEPHYR_TOOLCHAIN_VARIANT` and toolchain specific ones).

2.5 Build the Blinky Sample

Note: Blinky is compatible with most, but not all, boards. If your board does not meet Blinky's blinky-sample-requirements, then hello_world is a good alternative.

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

Ubuntu

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

macOS

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

Windows

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

The `-p auto` option automatically cleans byproducts from a previous build if necessary, which is useful if you try building another sample.

2.6 Flash the Sample

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

Then flash the sample using *west flash*:

```
west flash
```

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

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

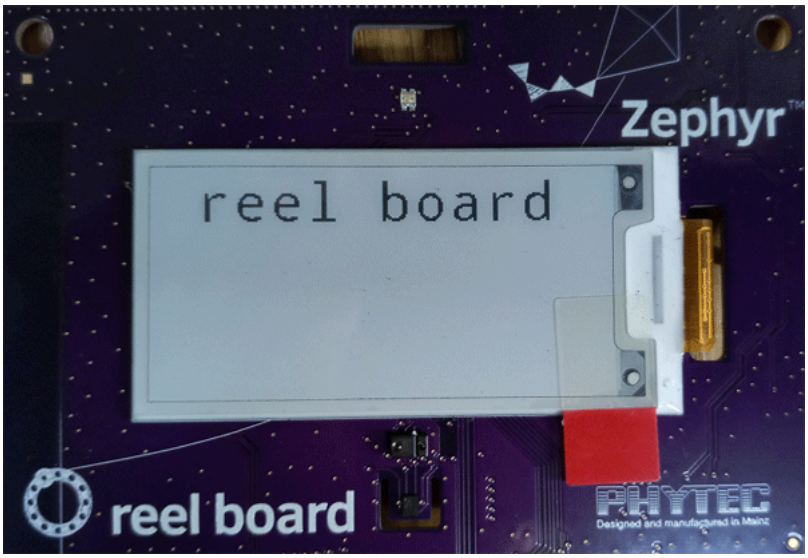


Fig. 1: Phytec reel_board running blinky

2.7 Next Steps

Here are some next steps for exploring Zephyr:

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

2.8 Asking for Help

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

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

2.8.1 How to Ask

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

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

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

2.8.2 Use Copy/Paste

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

Doing this makes it easier for people to help you, and also helps other users search the archives.

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

Chapter 3

Contribution Guidelines

As an open-source project, we welcome and encourage the community to submit patches directly to the project. In our collaborative open source environment, standards and methods for submitting changes help reduce the chaos that can result from an active development community.

This document explains how to participate in project conversations, log bugs and enhancement requests, and submit patches to the project so your patch will be accepted quickly in the codebase.

3.1 Licensing

Licensing is very important to open source projects. It helps ensure the software continues to be available under the terms that the author desired.

Zephyr uses the Apache 2.0 license (as found in the LICENSE file in the project's GitHub repo) to strike a balance between open contribution and allowing you to use the software however you would like to. The Apache 2.0 license is a permissive open source license that allows you to freely use, modify, distribute and sell your own products that include Apache 2.0 licensed software. (For more information about this, check out articles such as Why choose Apache 2.0 licensing and Top 10 Apache License Questions Answered).

A license tells you what rights you have as a developer, as provided by the copyright holder. It is important that the contributor fully understands the licensing rights and agrees to them. Sometimes the copyright holder isn't the contributor, such as when the contributor is doing work on behalf of a company.

3.1.1 Components using other Licenses

There are some imported or reused components of the Zephyr project that use other licensing, as described in *Licensing of Zephyr Project components*.

Importing code into the Zephyr OS from other projects that use a license other than the Apache 2.0 license needs to be fully understood in context and approved by the Zephyr governing board.

By carefully reviewing potential contributions and also enforcing a *Developer Certification of Origin (DCO)* for contributed code, we can ensure that the Zephyr community can develop products with the Zephyr Project without concerns over patent or copyright issues.

See *Contributing source code from external projects* for more information about this contributing and review process for imported components.

Licensing of Zephyr Project components

The Zephyr kernel tree imports or reuses packages, scripts and other files that are not covered by the Apache 2.0 License. In some places there is no LICENSE file or way to put a LICENSE file there, so we describe the licensing in this document.

scripts/{checkpatch.pl,checkstack.pl,get_maintainers.pl,spelling.txt} *Origin:* Linux Kernel

Licensing: GPLv2 License

3.2 Copyrights Notices

Please follow this Community Best Practice for Copyright Notices from the Linux Foundation.

3.3 Developer Certification of Origin (DCO)

To make a good faith effort to ensure licensing criteria are met, the Zephyr project requires the Developer Certificate of Origin (DCO) process to be followed.

The DCO is an attestation attached to every contribution made by every developer. In the commit message of the contribution, (described more fully later in this document), the developer simply adds a `Signed-off-by` statement and thereby agrees to the DCO.

When a developer submits a patch, it is a commitment that the contributor has the right to submit the patch per the license. The DCO agreement is shown below and at http://developercertificate.org/.

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the
    best of my knowledge, is covered under an appropriate open
    source license and I have the right under that license to
    submit that work with modifications, whether created in whole
    or in part by me, under the same open source license (unless
    I am permitted to submit under a different license), as
    Indicated in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including
    all personal information I submit with it, including my
    sign-off) is maintained indefinitely and may be redistributed
    consistent with this project or the open source license(s)
    involved.
```

3.3.1 DCO Sign-Off Methods

The DCO requires a sign-off message in the following format appear on each commit in the pull request:

```
Signed-off-by: Zephyrus Zephyr <zephyrus@zephyrproject.org>
```

The DCO text can either be manually added to your commit body, or you can add either `-s` or `--signoff` to your usual Git commit commands. If you forget to add the sign-off you can also amend a previous commit with the sign-off by running `git commit --amend -s`. If you've pushed your changes to GitHub already you'll need to force push your branch after this with `git push -f`.

3.3.2 Notes

Any contributions made as part of submitted pull requests are considered free for the Project to use. Developers are permitted to cherry-pick patches that are included in pull requests submitted by other contributors. It is expected that

- the content of the patches will not be substantially modified,
- the cherry-picked commits or portions of a commit shall preserve the original sign-off messages and the author identity.

Modifying Contributions made by other developers describes additional recommended policies around working with contributions submitted by other developers.

3.4 Prerequisites

As a contributor, you'll want to be familiar with the Zephyr project, how to configure, install, and use it as explained in the Zephyr Project website and how to set up your development environment as introduced in the Zephyr *Getting Started Guide*.

You should be familiar with common developer tools such as Git and CMake, and platforms such as GitHub.

If you haven't already done so, you'll need to create a (free) GitHub account on https://github.com and have Git tools available on your development system.

Note: The Zephyr development workflow supports all 3 major operating systems (Linux, macOS, and Windows) but some of the tools used in the sections below are only available on Linux and macOS. On Windows, instead of running these tools yourself, you will need to rely on the Continuous Integration (CI) service using Github Actions, which runs automatically on GitHub when you submit your Pull Request (PR). You can see any failure results in the workflow details link near the end of the PR conversation list. See *Continuous Integration* for more information

---

3.5 Repository layout

To clone the main Zephyr Project repositories use the instructions in *Get Zephyr and install Python dependencies*.

The Zephyr project directory structure is described in *Source Tree Structure* documentation. In addition to the Zephyr kernel itself, you'll also find the sources for technical documentation, sample code, supported board configurations, and a collection of subsystem tests. All of these are available for developers to contribute to and enhance.

3.6 Pull Requests and Issues

Before starting on a patch, first check in our issues Zephyr Project Issues system to see what's been reported on the issue you'd like to address. Have a conversation on the Zephyr devel mailing list (or the the Zephyr Discord Server) to see what others think of your issue (and proposed solution). You may find others that have encountered the issue you're finding, or that have similar ideas for changes or additions. Send a message to the Zephyr devel mailing list to introduce and discuss your idea with the development community.

It's always a good practice to search for existing or related issues before submitting your own. When you submit an issue (bug or feature request), the triage team will review and comment on the submission, typically within a few business days.

You can find all open pull requests on GitHub and open Zephyr Project Issues in Github issues.

3.7 Tools and Git Setup

3.7.1 Signed-off-by

The name in the commit message `Signed-off-by:` line and your email must match the change authorship information. Make sure your `.gitconfig` is set up correctly:

```
git config --global user.name "David Developer"
git config --global user.email "david.developer@company.com"
```

3.7.2 gitlint

When you submit a pull request to the project, a series of checks are performed to verify your commit messages meet the requirements. The same step done during the CI process can be performed locally using the the `gitlint` command.

Run `gitlint` locally in your tree and branch where your patches have been committed:

```
gitlint
```

Note, gitlint only checks HEAD (the most recent commit), so you should run it after each commit, or use the `--commits` option to specify a commit range covering all the development patches to be submitted.

3.7.3 twister

Note: twister does not currently run on Windows.

To verify that your changes did not break any tests or samples, please run the `twister` script locally before submitting your pull request to GitHub. To run the same tests the CI system runs, follow these steps from within your local Zephyr source working directory:

```
source zephyr-env.sh
./scripts/twister
```

The above will execute the basic twister script, which will run various kernel tests using the QEMU emulator. It will also do some build tests on various samples with advanced features that can't run in QEMU.

We highly recommend you run these tests locally to avoid any CI failures.

3.7.4 uncrustify

The uncrustify tool can be helpful to quickly reformat large amounts of new source code to our *Coding Style* standards together with a configuration file we've provided:

```
# On Linux/macOS
uncrustify --replace --no-backup -l C -c $ZEPHYR_BASE/.uncrustify.cfg my_source_file.c
# On Windows
uncrustify --replace --no-backup -l C -c %ZEPHYR_BASE%\.uncrustify.cfg my_source_file.
↪c
```

But note that you should not use uncrustify to reformat existing Zephyr code, or to modify files in which you only introduce a small fix. This would create a lot of unwelcome extra changed lines.

On Linux systems, you can install uncrustify with

```
sudo apt install uncrustify
```

For Windows installation instructions see the sourceforge listing for uncrustify.

3.8 Coding Style

Use these coding guidelines to ensure that your development complies with the project's style and naming conventions.

In general, follow the Linux kernel coding style, with the following exceptions:

- Add braces to every `if`, `else`, `do`, `while`, `for` and `switch` body, even for single-line code blocks. Use the `--ignore BRACES` flag to make *checkpatch* stop complaining.
- Use spaces instead of tabs to align comments after declarations, as needed.
- Use C89-style single line comments, `/* */`. The C99-style single line comment, `//`, is not allowed.
- Use `/** */` for doxygen comments that need to appear in the documentation.

The Linux kernel GPL-licensed tool `checkpatch` is used to check coding style conformity.

Note: checkpatch does not currently run on Windows.

Checkpatch is available in the scripts directory. To invoke it when committing code, make the file *$ZEPHYR_BASE/.git/hooks/pre-commit* executable and edit it to contain:

```
#!/bin/sh
set -e exec
exec git diff --cached | ${ZEPHYR_BASE}/scripts/checkpatch.pl -
```

Instead of running checkpatch at each commit, you may prefer to run it only before pushing on zephyr repo. To do this, make the file *$ZEPHYR_BASE/.git/hooks/pre-push* executable and edit it to contain:

```
#!/bin/sh
remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000
```

(continues on next page)

(continued from previous page)

```
echo "Run push hook"

while read local_ref local_sha remote_ref remote_sha
do
    args="$remote $url $local_ref $local_sha $remote_ref $remote_sha"
    exec ${ZEPHYR_BASE}/scripts/series-push-hook.sh $args
done

exit 0
```

If you want to override checkpatch verdict and push you branch despite reported issues, you can add option –no-verify to the git push command.

A more complete alternative to this is using check_compliance.py script from ci-tools repo.

3.9 Other Guidelines

Beyond the *Coding Style* that Zephyr enforces for all code that is submitted for inclusion, the project targets compliance with a series of coding guidelines. Refer to the *Coding Guidelines* section of the documentation for additional details.

3.9.1 Coding Guidelines

The project TSC and the Safety Committee of the project agreed to implement a staged and incremental approach for complying with a set of coding rules (AKA Coding Guidelines) to improve quality and consistency of the code base. Below are the agreed upon stages and the approximate timelines:

Stage I Coding guideline rules are available to be followed and referenced, but not enforced. Rules are not yet enforced in CI and pull-requests cannot be blocked by reviewers/approvers due to violations.

Stage II Begin enforcement on a limited scope of the code base. Initially, this would be the safety certification scope. For rules easily applied across codebase, we should not limit compliance to initial scope. This step requires tooling and CI setup and will start sometime after LTS2.

Stage III Revisit the coding guideline rules and based on experience from previous stages, refine/iterate on selected rules.

Stage IV Expand enforcement to the wider codebase. Exceptions may be granted on some areas of the codebase with a proper justification. Exception would require TSC approval.

Note: Coding guideline rules may be removed/changed at any time by filing a GH issue/RFC.

Main rules

The coding guideline rules are based on MISRA-C 2012 and are a subset of MISRA-C. The subset is listed in the table below with a summary of the rules, its severity and the equivlent rules from other standards for reference.

Note: For existing Zephyr maintainers and collaborators, if you are unable to obtain a copy through your employer, a limited number of copies will be made available through the project. If you need a copy of MISRA-C 2012, please send email to safety@lists.zephyrproject.org and provide details on reason why

you can't obtain one through other options and expected contributions once you have one. The safety committee will review all requests.

Table 1: Main rules

MISRA C 2012	Severity	Description	CERT C	Example
Dir 1.1	Required	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood	MSC09-C	Dir 1.1
Dir 2.1	Required	All source files shall compile without any compilation errors	N/A	Dir 2.1
Dir 3.1	Required	All code shall be traceable to documented requirements	N/A	Dir 3.1
Dir 4.1	Required	Run-time failures shall be minimized	N/A	Dir 4.1
Dir 4.2	Advisory	All usage of assembly language should be documented	N/A	Dir 4.2
Dir 4.4	Advisory	Sections of code should not be "commented out"	MSC04-C	Dir 4.4
Dir 4.5	Advisory	Identifiers in the same name space with overlapping visibility should be typographically unambiguous	DCL02-C	Dir 4.5
Dir 4.6	Advisory	typedefs that indicate size and signedness should be used in place of the basic numerical types	N/A	Dir 4.6
Dir 4.7	Required	If a function returns error information, then that error information shall be tested	N/A	Dir 4.7

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Dir 4.8	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden	DCL12-C	Dir 4.8 example 1 Dir 4.8 example 2
Dir 4.9	Advisory	A function should be used in preference to a function-like macro where they are interchangeable	PRE00-C	Dir 4.9
Dir 4.10	Required	Precautions shall be taken in order to prevent the contents of a header file being included more than once	PRE06-C	Dir 4.10
Dir 4.11	Required	The validity of values passed to library functions shall be checked	N/A	Dir 4.11
Dir 4.12	Required	Dynamic memory allocation shall not be used	STR01-C	Dir 4.12
Dir 4.13	Advisory	Functions which are designed to provide operations on a resource should be called in an appropriate sequence	N/A	Dir 4.13
Dir 4.14	Required	The validity of values received from external sources shall be checked	N/A	Dir 4.14
Rule 1.2	Advisory	Language extensions should not be used	MSC04-C	Rule 1.2
Rule 1.3	Required	There shall be no occurrence of undefined or critical unspecified behaviour	N/A	Rule 1.3

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 2.1	Required	A project shall not contain unreachable code	MSC07-C	Rule 2.1 example 1 Rule 2.1 example 2
Rule 2.2	Required	There shall be no dead code	MSC12-C	Rule 2.2
Rule 2.3	Advisory	A project should not contain unused type declarations	N/A	Rule 2.3
Rule 2.6	Advisory	A function should not contain unused label declarations	N/A	Rule 2.6
Rule 2.7	Advisory	There should be no unused parameters in functions	N/A	Rule 2.7
Rule 3.1	Required	The character sequences /* and // shall not be used within a comment	MSC04-C	Rule 3.1
Rule 3.2	Required	Line-splicing shall not be used in // comments	N/A	Rule 3.2
Rule 4.1	Required	Octal and hexadecimal escape sequences shall be terminated	MSC09-C	Rule 4.1
Rule 4.2	Advisory	Trigraphs should not be used	PRE07-C	Rule 4.2
Rule 5.1	Required	External identifiers shall be distinct	DCL23-C	Rule 5.1 example 1 Rule 5.1 example 2
Rule 5.2	Required	Identifiers declared in the same scope and name space shall be distinct	DCL23-C	Rule 5.2
Rule 5.3	Required	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	DCL23-C	Rule 5.3
Rule 5.4	Required	Macro identifiers shall be distinct	DCL23-C	Rule 5.4
Rule 5.5	Required	Identifiers shall be distinct from macro names	DCL23-C	Rule 5.5

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 5.6	Required	A typedef name shall be a unique identifier	N/A	Rule 5.6
Rule 5.7	Required	A tag name shall be a unique identifier	N/A	Rule 5.7
Rule 5.8	Required	Identifiers that define objects or functions with external linkage shall be unique	N/A	Rule 5.8 example 1 Rule 5.8 example 2
Rule 5.9	Advisory	Identifiers that define objects or functions with internal linkage should be unique	N/A	Rule 5.9 example 1 Rule 5.9 example 2
Rule 6.1	Required	Bit-fields shall only be declared with an appropriate type	INT14-C	Rule 6.1
Rule 6.2	Required	Single-bit named bit fields shall not be of a signed type	INT14-C	Rule 6.2
Rule 7.1	Required	Octal constants shall not be used	DCL18-C	Rule 7.1
Rule 7.2	Required	A u or U suffix shall be applied to all integer constants that are represented in an unsigned type	N/A	Rule 7.2
Rule 7.3	Required	The lowercase character l shall not be used in a literal suffix	DCL16-C	Rule 7.3
Rule 7.4	Required	A string literal shall not be assigned to an object unless the objects type is pointer to const-qualified char	N/A	Rule 7.4
Rule 8.1	Required	Types shall be explicitly specified	N/A	Rule 8.1
Rule 8.2	Required	Function types shall be in prototype form with named parameters	DCL20-C	Rule 8.2

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 8.3	Required	All declarations of an object or function shall use the same names and type qualifiers	N/A	Rule 8.3
Rule 8.4	Required	A compatible declaration shall be visible when an object or function with external linkage is defined	N/A	Rule 8.4
Rule 8.5	Required	An external object or function shall be declared once in one and only one file	N/A	Rule 8.5 example 1 Rule 8.5 example 2
Rule 8.6	Required	An identifier with external linkage shall have exactly one external definition	N/A	Rule 8.6 example 1 Rule 8.6 example 2
Rule 8.8	Required	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	DCL15-C	Rule 8.8
Rule 8.9	Advisory	An object should be defined at block scope if its identifier only appears in a single function	DCL19-C	Rule 8.9
Rule 8.10	Required	An inline function shall be declared with the static storage class	N/A	Rule 8.10
Rule 8.12	Required	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique	INT09-C	Rule 8.12
Rule 8.14	Required	The restrict type qualifier shall not be used	N/A	Rule 8.14

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 9.1	Mandatory	The value of an object with automatic storage duration shall not be read before it has been set	N/A	Rule 9.1
Rule 9.2	Required	The initializer for an aggregate or union shall be enclosed in braces	N/A	Rule 9.2
Rule 9.3	Required	Arrays shall not be partially initialized	N/A	Rule 9.3
Rule 9.4	Required	An element of an object shall not be initialized more than once	N/A	Rule 9.4
Rule 9.5	Required	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly	N/A	Rule 9.5
Rule 10.1	Required	Operands shall not be of an inappropriate essential type	STR04-C	Rule 10.1
Rule 10.2	Required	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	STR04-C	Rule 10.2
Rule 10.3	Required	The value of an expression shall not be assigned to an object with a narrower essential type or of a dierent essential type category	STR04-C	Rule 10.3
Rule 10.4	Required	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	STR04-C	Rule 10.4

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 10.5	Advisory	The value of an expression should not be cast to an inappropriate essential type	N/A	Rule 10.5
Rule 10.6	Required	The value of a composite expression shall not be assigned to an object with wider essential type	INT02-C	Rule 10.6
Rule 10.7	Required	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	INT02-C	Rule 10.7
Rule 10.8	Required	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	INT02-C	Rule 10.8
Rule 11.2	Required	Conversions shall not be performed between a pointer to an incomplete type and any other type	N/A	Rule 11.2
Rule 11.6	Required	A cast shall not be performed between pointer to void and an arithmetic type	N/A	Rule 11.6
Rule 11.7	Required	A cast shall not be performed between pointer to object and a non-integer arithmetic type	N/A	Rule 11.7
Rule 11.8	Required	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer	EXP05-C	Rule 11.8

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 11.9	Required	The macro NULL shall be the only permitted form of integer null pointer constant	N/A	Rule 11.9
Rule 12.1	Advisory	The precedence of operators within expressions should be made explicit	EXP00-C	Rule 12.1
Rule 12.2	Required	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand	N/A	Rule 12.2
Rule 12.4	Advisory	Evaluation of constant expressions should not lead to unsigned integer wrap-around	N/A	Rule 12.4
Rule 12.5	Mandatory	The sizeof operator shall not have an operand which is a function parameter declared as “array of type”	N/A	Rule 12.5
Rule 13.1	Required	Initializer lists shall not contain persistent side effects	N/A	Rule 13.1 example 1 Rule 13.1 example 2
Rule 13.2	Required	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	N/A	Rule 13.2
Rule 13.3	Advisory	A full expression containing an increment (++) or decrement (–) operator should have no other potential side effects other than that caused by the increment or decrement operator	N/A	Rule 13.3

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 13.4	Advisory	The result of an assignment operator should not be used	N/A	Rule 13.4
Rule 13.5	Required	The right hand operand of a logical && or || operator shall not contain persistent side effects	EXP10-C	Rule 13.5 example 1 Rule 13.5 example 2
Rule 13.6	Mandatory	The operand of the sizeof operator shall not contain any expression which has potential side effects	N/A	Rule 13.6
Rule 14.1	Required	A loop counter shall not have essentially floating type	N/A	Rule 14.1
Rule 14.2	Required	A for loop shall be well-formed	N/A	Rule 14.2
Rule 14.3	Required	Controlling expressions shall not be invariant	N/A	Rule 14.3
Rule 14.4	Required	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type	N/A	Rule 14.4
Rule 15.2	Required	The goto statement shall jump to a label declared later in the same function	N/A	Rule 15.2
Rule 15.3	Required	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	N/A	Rule 15.3
Rule 15.6	Required	The body of an iteration-statement or a selection-statement shall be a compound-statement	EXP19-C	Rule 15.6

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 15.7	Required	All if else if constructs shall be terminated with an else statement	N/A	Rule 15.7
Rule 16.1	Required	All switch statements shall be well-formed	N/A	Rule 16.1
Rule 16.2	Required	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	MSC20-C	Rule 16.2
Rule 16.3	Required	An unconditional break statement shall terminate every switch-clause	N/A	Rule 16.3
Rule 16.4	Required	Every switch statement shall have a default label	N/A	Rule 16.4
Rule 16.5	Required	A default label shall appear as either the first or the last switch label of a switch statement	N/A	Rule 16.5
Rule 16.6	Required	Every switch statement shall have at least two switch-clauses	N/A	Rule 16.6
Rule 16.7	Required	A switch-expression shall not have essentially Boolean type	N/A	Rule 16.7
Rule 17.1	Required	The features of <stdarg.h> shall not be used	ERR00-C	Rule 17.1
Rule 17.2	Required	Functions shall not call themselves, either directly or indirectly	MEM05-C	Rule 17.2
Rule 17.3	Mandatory	A function shall not be declared implicitly	N/A	Rule 17.3

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 17.4	Mandatory	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	N/A	Rule 17.4
Rule 17.5	Advisory	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements	N/A	Rule 17.5
Rule 17.6	Mandatory	The declaration of an array parameter shall not contain the static keyword between the [ ]	N/A	Rule 17.6
Rule 17.7	Required	The value returned by a function having non-void return type shall be used	N/A	Rule 17.7
Rule 18.1	Required	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand	EXP08-C	Rule 18.1
Rule 18.2	Required	Subtraction between pointers shall only be applied to pointers that address elements of the same array	EXP08-C	Rule 18.2
Rule 18.3	Required	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object	EXP08-C	Rule 18.3

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 18.5	Advisory	Declarations should contain no more than two levels of pointer nesting	N/A	Rule 18.5
Rule 18.6	Required	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist	N/A	Rule 18.6 example 1 Rule 18.6 example 2
Rule 18.8	Required	Variable-length array types shall not be used	N/A	Rule 18.8
Rule 19.1	Mandatory	An object shall not be assigned or copied to an overlapping object	N/A	Rule 19.1
Rule 20.2	Required	The ‘, or characters and the /* or // character sequences shall not occur in a header file name”	N/A	Rule 20.2
Rule 20.3	Required	The #include directive shall be followed by either a <filename> or “filename” sequence	N/A	Rule 20.3
Rule 20.4	Required	A macro shall not be defined with the same name as a keyword	N/A	Rule 20.4
Rule 20.7	Required	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	PRE01-C	Rule 20.7
Rule 20.8	Required	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	N/A	Rule 20.8

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 20.9	Required	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #defined before evaluation	N/A	Rule 20.9
Rule 20.11	Required	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	N/A	Rule 20.11
Rule 20.12	Required	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators	N/A	Rule 20.12
Rule 20.13	Required	A line whose first token is # shall be a valid preprocessing directive	N/A	Rule 20.13
Rule 20.14	Required	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	N/A	Rule 20.14
Rule 21.1	Required	#define and #undef shall not be used on a reserved identifier or reserved macro name	N/A	Rule 21.1
Rule 21.2	Required	A reserved identifier or macro name shall not be declared	N/A	Rule 21.2
Rule 21.3	Required	The memory allocation and deallocation functions of <stdlib.h> shall not be used	MSC24-C	Rule 21.3

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 21.4	Required	The standard header file <setjmp.h> shall not be used	N/A	Rule 21.4
Rule 21.6	Required	The Standard Library input/output functions shall not be used	N/A	Rule 21.6
Rule 21.7	Required	The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used	N/A	Rule 21.7
Rule 21.9	Required	The library functions bsearch and qsort of <stdlib.h> shall not be used	N/A	Rule 21.9
Rule 21.11	Required	The standard header file <tgmath.h> shall not be used	N/A	Rule 21.11
Rule 21.12	Advisory	The exception handling features of <fenv.h> should not be used	N/A	Rule 21.12
Rule 21.13	Mandatory	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EO	N/A	Rule 21.13
Rule 21.14	Required	The Standard Library function memcmp shall not be used to compare null terminated strings	N/A	Rule 21.14
Rule 21.15	Required	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types	N/A	Rule 21.15

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 21.16	Required	The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type	N/A	Rule 21.16
Rule 21.17	Mandatory	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	N/A	Rule 21.17
Rule 21.18	Mandatory	The size_t argument passed to any function in <string.h> shall have an appropriate value	N/A	Rule 21.18
Rule 21.19	Mandatory	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type	N/A	Rule 21.19
Rule 21.20	Mandatory	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function	N/A	Rule 21.20

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 22.1	Required	All resources obtained dynamically by means of Standard Library functions shall be explicitly released	N/A	Rule 22.1
Rule 22.2	Mandatory	A block of memory shall only be freed if it was allocated by means of a Standard Library function	N/A	Rule 22.2
Rule 22.3	Required	The same file shall not be open for read and write access at the same time on different streams	N/A	Rule 22.3
Rule 22.4	Mandatory	There shall be no attempt to write to a stream which has been opened as read-only	N/A	Rule 22.4
Rule 22.5	Mandatory	A pointer to a FILE object shall not be dereferenced	N/A	Rule 22.5
Rule 22.6	Mandatory	The value of a pointer to a FILE shall not be used after the associated stream has been closed	N/A	Rule 22.6
Rule 22.7	Required	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF	N/A	Rule 22.7
Rule 22.8	Required	The value of errno shall be set to zero prior to a call to an errno-setting-function	N/A	Rule 22.8
Rule 22.9	Required	The value of errno shall be tested against zero after calling an errno-setting-function	N/A	Rule 22.9

continues on next page

Table 1 – continued from previous page

MISRA C 2012	Severity	Description	CERT C	Example
Rule 22.10	Required	The value of errno shall only be tested when the last function to be called was an errno-setting-function	N/A	Rule 22.10

Additional rules

Rule A.1: Conditional Compilation

Severity Required

Description Do not conditionally compile function declarations in header files. Do not conditionally compile structure declarations in header files. You may conditionally exclude fields within structure definitions to avoid wasting memory when the feature they support is not enabled.

Rationale Excluding declarations from the header based on compile-time options may prevent their documentation from being generated. Their absence also prevents use of `if (IS_ENABLED(CONFIG_FOO)) {}` as an alternative to preprocessor conditionals when the code path should change based on the selected options.

Rule A.2: Inclusive Language

Severity Required

Description Do not introduce new usage of offensive terms listed below. This rule applies but is not limited to source code, comments, documentation, and branch names. Replacement terms may vary by area or subsystem, but should aim to follow updated industry standards when possible.

Exceptions are allowed for maintaining existing implementations or adding new implementations of industry standard specifications governed externally to the Zephyr Project.

Existing usage is recommended to change as soon as updated industry standard specifications become available or new terms are publicly announced by the governing body, or immediately if no specifications apply.

Offensive Terms	Recommended Replacements
`{master,leader} / slave`	• `{primary,main} / {secondary, replica}` • `{initiator,requester} / {target, responder}` • `{controller,host} / {device,worker, proxy,target}` • `director / performer` • `central / peripheral`
`blacklist / whitelist`	• `denylist / allowlist` • `blocklist / allowlist` • `rejectlist / acceptlist`
`grandfather policy`	• `legacy`
`sanity`	• `coherence` • `confidence`

Rationale Offensive terms do not create an inclusive community environment and therefore violate the Zephyr Project Code of Conduct. This coding rule was inspired by a similar rule in Linux.

Status Related GitHub Issues and Pull Requests are tagged with the Inclusive Language Label.

Area	Selected Replacements	Status
Bluetooth	See Bluetooth Appropriate Language Mapping Tables	
eSPI	• `master / slave` => TBD	
gPTP	• `master / slave` => TBD	
I2C	• `master / slave` => TBD	NXP publishes the I2C Specification and has selected `controller / target` as replacement terms, but the timing to publish an announcement or new specification is TBD. Zephyr will update I2C when replacement terminology is confirmed by a public announcement or updated specification. See Zephyr issue 27033.
I2S	• `master / slave` => TBD	
SMP/AMP	• `master / slave` => TBD	
SPI	• `master / slave` => `controller / peripheral` • `MOSI / MISO / SS` => `SDO / SDI / CS`	The Open Source Hardware Association has selected these replacement terms. See OSHWA Resolution to Redefine SPI Signal Names
Test Runner (Twister)	• `platform_whitelist` => `platform_allow` • `sanitycheck` => `twister`	

3.9.2 Documentation Guidelines

Note: For instructions on building the documentation, see *Documentation Generation*.

Zephyr Project content is written using the reStructuredText markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted standalone website. Developers can view this content either in its raw form as .rst markup files, or (with Sphinx installed) they can build the documentation using the Makefile on Linux systems, or make.bat on Windows, to generate the HTML content. The HTML content can then be viewed using a web browser. This same .rst content is also fed into the Zephyr documentation website (with a different theme applied).

You can read details about reStructuredText and about Sphinx extensions from their respective websites.

This document provides a quick reference for commonly used reST and Sphinx-defined directives and roles used to create the documentation you're reading.

Headings

While reST allows use of both and overline and matching underline to indicate a heading, we only use an underline indicator for headings.

- Document title (h1) use "#" for the underline character
- First section heading level (h2) use "*"
- Second section heading level (h3) use "="
- Third section heading level (h4) use "-"

The heading underline must be at least as long as the title it's under.

For example:

```
This is a title heading
#######################

some content goes here

First section heading
*********************
```

Content Highlighting

Some common reST inline markup samples:

- one asterisk: `*text*` for emphasis (*italics*),
- two asterisks: `**text**` for strong emphasis (**boldface**), and
- two backquotes: ``` ``text`` ``` for `inline code` samples.

If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, you can eliminate the confusion by adding a backslash (\) before it.

Lists

For bullet lists, place an asterisk (*) or hyphen (-) at the start of a paragraph and indent continuation lines with two spaces.

The first item in a list (or sublist) must have a blank line before it and should be indented at the same level as the preceding paragraph (and not indented itself).

For numbered lists start with a 1. or a. for example, and continue with autonumbering by using a # sign. Indent continuation lines with three spaces:

```
* This is a bulleted list.
* It has two items, the second
  item and has more than one line of reST text.  Additional lines
  are indented to the first character of the
  text of the bullet list.

1. This is a new numbered list. If the wasn't a blank line before it,
   it would be a continuation of the previous list (or paragraph).
#. It has two items too.

a. This is a numbered list using alphabetic list headings
#. It has three items (and uses autonumbering for the rest of the list)
```

(continues on next page)

(continued from previous page)

```
#. Here's the third item

#. This is an autonumbered list (default is to use numbers starting
   with 1).

   #. This is a second-level list under the first item (also
      autonumbered).  Notice the indenting.
   #. And a second item in the nested list.
#. And a second item back in the containing list.  No blank line
   needed, but it wouldn't hurt for readability.
```

Definition lists (with a term and its definition) are a convenient way to document a word or phrase with an explanation. For example this reST content:

```
The Makefile has targets that include:

html
   Build the HTML output for the project

clean
   Remove all generated output, restoring the folders to a
   clean state.
```

Would be rendered as:

The Makefile has targets that include:

html Build the HTML output for the project

clean Remove all generated output, restoring the folders to a clean state.

Multi-column lists

If you have a long bullet list of items, where each item is short, you can indicate the list items should be rendered in multiple columns with a special `.. rst-class:: rst-columns` directive. The directive will apply to the next non-comment element (e.g., paragraph), or to content indented under the directive. For example, this unordered list:

```
.. rst-class:: rst-columns

* A list of
* short items
* that should be
* displayed
* horizontally
* so it doesn't
* use up so much
* space on
* the page
```

would be rendered as:

- A list of
- short items
- that should be
- displayed
- horizontally

- so it doesn't
- use up so much
- space on
- the page

A maximum of three columns will be displayed, and change based on the available width of the display window, reducing to one column on narrow (phone) screens if necessary. We've deprecated use of the `hlist` directive because it misbehaves on smaller screens.

Tables

There are a few ways to create tables, each with their limitations or quirks. Grid tables offer the most capability for defining merged rows and columns, but are hard to maintain:

```
+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | ...        | ...      | you can  |
+------------------------+------------+----------+ easily   +
| body row 3 with a two column span   | ...      | span     |
+------------------------+------------+----------+ rows     +
| body row 4             | ...        | ...      | too      |
+------------------------+------------+----------+----------+
```

This example would render as:

Header row, column 1 (header rows optional)	Header 2	Header 3	Header 4
body row 1, column 1	column 2	column 3	column 4
body row 2	...	...	you can easily span rows too
body row 3 with a two column span		...	
body row 4	...	...	

List tables are much easier to maintain, but don't support row or column spans:

```
.. list-table:: Table title
   :widths: 15 20 40
   :header-rows: 1

   * - Heading 1
     - Heading 2
     - Heading 3
   * - body row 1, column 1
     - body row 1, column 2
     - body row 1, column 3
   * - body row 2, column 1
     - body row 2, column 2
     - body row 2, column 3
```

This example would render as:

Table 2: Table title

Heading 1	Heading 2	Heading 3
body row 1, column 1	body row 1, column 2	body row 1, column 3
body row 2, column 1	body row 2, column 2	body row 2, column 3

The `:widths:` parameter lets you define relative column widths. The default is equal column widths. If you have a three-column table and you want the first column to be half as wide as the other two equal-width columns, you can specify `:widths: 1 2 2`. If you'd like the browser to set the column widths automatically based on the column contents, you can use `:widths: auto`.

File names and Commands

Sphinx extends reST by supporting additional inline markup elements (called "roles") used to tag text with special meanings and allow style output formatting. (You can refer to the Sphinx Inline Markup documentation for the full list).

For example, there are roles for marking `filenames` (`:file:`name``) and command names such as `make` (`:command:`make``). You can also use the ``inline code`` markup (double backticks) to indicate a `filename`.

For references to files that are in the Zephyr GitHub tree, a special role can be used that creates a hyperlink to that file. For example a reference to the reST file used to create this document can be generated using `:zephyr_file:\`doc/guides/documentation/index.rst\`` that will show up as doc/guides/documentation/index.rst, a link to the "blob" file in the github repo. There's also a `:zephyr_raw:\`doc/guides/documentation/index.rst\`` role that will link to the "raw" content, doc/guides/documentation/index.rst. (You can click on these links to see the difference.)

Internal Cross-Reference Linking

Traditional ReST links are only supported within the current file using the notation:

```
Refer to the `internal-linking`_ page
```

which renders as,

> Refer to the *internal-linking* page

Note the use of a trailing underscore to indicate an outbound link. In this example, the label was added immediately before a heading, so the text that's displayed is the heading text itself. You can change the text that's displayed as the link writing this as:

```
Refer to the `show this text instead <internal-linking>`_ page
```

which renders as,

> Refer to the show this text instead page

External Cross-Reference Linking

With Sphinx's help, we can create link-references to any tagged text within the Zephyr Project documentation.

Target locations in a document are defined with a label directive:

```
.. _my label name:

Heading
=======
```

Note the leading underscore indicating an inbound link. The content immediately following this label must be a heading, and is the target for a `:ref:`my label name`` reference from anywhere within the Zephyr documentation. The heading text is shown when referencing this label. You can also change the text that's displayed for this link, such as:

```
:ref:`some other text <my label name>`
```

To enable easy cross-page linking within the site, each file should have a reference label before its title so it can be referenced from another file. These reference labels must be unique across the whole site, so generic names such as "samples" should be avoided. For example the top of this document's .rst file is:

```
.. _doc_guidelines:

Documentation Guidelines for the Zephyr Project
###############################################
```

Other .rst documents can link to this document using the `:ref:`doc_guidelines`` tag and it will show up as *Documentation Guidelines*. This type of internal cross reference works across multiple files, and the link text is obtained from the document source so if the title changes, the link text will update as well.

You can also define links to any URL and then reference it in your document. For example, with this label definition in the document:

```
.. _Zephyr Wikipedia Page:
   https://en.wikipedia.org/wiki/Zephyr_(operating_system)
```

you can reference it with:

```
Read the `Zephyr Wikipedia Page`_ for more information about the
project.
```

`any` links

Within the Zephyr project, we've defined the default role to be "any", meaning if you just write a phrase in back-ticks, e.g., `` `doc_guidelines` ``, Sphinx will search through all domains looking for something called doc_guidelines to link to. In this case it will find the label at the top of this document, and link to *Documentation Guidelines*. This can be useful for linking to doxygen-generated links for function names and such, but will cause a warning such as:

```
WARNING: 'any' reference target not found: doc_giudelines
```

if you misspelled `` `doc_guidelines` `` as `` `doc_giudelines` ``.

Non-ASCII Characters

You can insert non-ASCII characters such as a Trademark symbol (™), by using the notation `|trade|`. Available replacement names are defined in an include file used during the Sphinx processing of the reST files. The names of these replacement characters are the same as used in HTML entities used to insert characters in HTML, e.g., ™ and are defined in the file `sphinx_build/substitutions.txt` as listed here:

```
.. |br|     raw:: html        .. force a line break in HTML output (blank lines␣
↪needed here)

   <br />

.. |p|     raw:: html        .. force a blank line in HTML output (blank lines needed␣
↪here)

   <p></p>

.. These are replacement strings for non-ASCII characters used within the project
   using the same name as the html entity names (e.g., &copy;) for that character

.. |copy|   unicode:: U+000A9 .. COPYRIGHT SIGN
   :ltrim:
.. |trade|  unicode:: U+02122 .. TRADEMARK SIGN
   :ltrim:
.. |reg|    unicode:: U+000AE .. REGISTERED TRADEMARK SIGN
   :ltrim:
.. |deg|    unicode:: U+000B0 .. DEGREE SIGN
   :ltrim:
.. |plusminus|  unicode:: U+000B1 .. PLUS-MINUS SIGN
   :rtrim:
.. |micro|  unicode:: U+000B5 .. MICRO SIGN
   :rtrim:
.. |sup2|  unicode:: U+00B2 .. SUPERSCRIPT TWO
   :ltrim:
```

We've kept the substitutions list small but others can be added as needed by submitting a change to the `substitutions.txt` file.

Code and Command Examples

Use the reST `code-block` directive to create a highlighted block of fixed-width text, typically used for showing formatted code or console commands and output. Smart syntax highlighting is also supported (using the Pygments package). You can also directly specify the highlighting language. For example:

```
.. code-block:: c

   struct z_object {
      char *name;
      uint8_t perms[CONFIG_MAX_THREAD_BYTES];
      uint8_t type;
      uint8_t flags;
      uint32_t data;
   } __packed;
```

Note the blank line between the `code-block` directive and the first line of the code-block body, and the body content is indented three spaces (to the first non-white space of the directive name).

This would be rendered as:

```
struct z_object {
   char *name;
   uint8_t perms[CONFIG_MAX_THREAD_BYTES];
   uint8_t type;
   uint8_t flags;
```

(continues on next page)

(continued from previous page)

```
   uint32_t data;
} __packed;
```

You can specify other languages for the `code-block` directive, including `c`, `python`, and `rst`, and also `console`, `bash`, or `shell`. If you want no syntax highlighting, use the language `none`, for example:

```
.. code-block:: none

   This would be a block of text styled with a background
   and box, but with no syntax highlighting.
```

Would display as:

```
This would be a block of text styled with a background
and box, but with no syntax highlighting.
```

There's a shorthand for writing code blocks too: end the introductory paragraph with a double colon (`::`) and indent the code block content by three spaces. On output, only one colon will be shown. The highlighting package makes a best guess at the type of content in the block and highlighting purposes.

Images

Images are included in documentation by using an image directive:

```
.. image:: ../../../../images/doc-gen-flow.png
   :align: center
   :alt: alt text for the image
```

or if you'd like to add an image caption, use:

```
.. figure:: ../../../../images/doc-gen-flow.png
   :alt: image description

   Caption for the figure
```

The file name specified is relative to the document source file, and we recommend putting images into an `images` folder where the document source is found. The usual image formats handled by a web browser are supported: JPEG, PNG, GIF, and SVG. Keep the image size only as large as needed, generally at least 500 px wide but no more than 1000 px, and no more than 250 KB unless a particularly large image is needed for clarity.

Tabs, spaces, and indenting

Indenting is significant in reST file content, and using spaces is preferred. Extra indenting can (unintentionally) change the way content is rendered too. For lists and directives, indent the content text to the first non-white space in the preceding line. For example:

```
* List item that spans multiple lines of text
  showing where to indent the continuation line.

1. And for numbered list items, the continuation
   line should align with the text of the line above.

.. code-block::
```

(continues on next page)

(continued from previous page)

```
   The text within a directive block should align with the
   first character of the directive name.
```

Keep the line length for documentation less than 80 characters to make it easier for reviewing in GitHub. Long lines because of URL references are an allowed exception.

zephyr-app-commands Directive

This is a Zephyr directive for generating consistent documentation of the shell commands needed to manage (build, flash, etc.) an application.

For example, to generate commands to build samples/hello_world for qemu_x86 use:

```
.. zephyr-app-commands::
   :zephyr-app: samples/hello_world
   :board: qemu_x86
   :goals: build
```

Directive options:

:tool: which tool to use. Valid options are currently 'cmake', 'west' and 'all'. The default is 'west'.

:app: path to the application to build.

:zephyr-app: path to the application to build, this is an app present in the upstream zephyr repository. Mutually exclusive with :app:.

:cd-into: if set, build instructions are given from within the :app: folder, instead of outside of it.

:generator: which build system to generate. Valid options are currently 'ninja' and 'make'. The default is 'ninja'. This option is not case sensitive.

:host-os: which host OS the instructions are for. Valid options are 'unix', 'win' and 'all'. The default is 'all'.

:board: if set, the application build will target the given board.

:shield: if set, the application build will target the given shield.

:conf: if set, the application build will use the given configuration file. If multiple conf files are provided, enclose the space-separated list of files with quotes, e.g., "a.conf b.conf".

:gen-args: if set, additional arguments to the CMake invocation

:build-args: if set, additional arguments to the build invocation

:build-dir: if set, the application build directory will *APPEND* this (relative, Unix-separated) path to the standard build directory. This is mostly useful for distinguishing builds for one application within a single page.

:goals: a whitespace-separated list of what to do with the app (in 'build', 'flash', 'debug', 'debugserver', 'run'). Commands to accomplish these tasks will be generated in the right order.

:maybe-skip-config: if set, this indicates the reader may have already created a build directory and changed there, and will tweak the text to note that doing so again is not necessary.

:compact: if set, the generated output is a single code block with no additional comment lines

For example, the `.. zephyr-app-commands` listed above would render like this in the generated HTML output:

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

Alternative Tabbed Content

As introduced in the *Getting Started Guide*, you can provide alternative content to the reader via a tabbed interface. When the reader clicks on a tab, the content for that tab is displayed, for example:

```
.. tabs::

   .. tab:: Apples

      Apples are green, or sometimes red.

   .. tab:: Pears

      Pears are green.

   .. tab:: Oranges

      Oranges are orange.
```

will display as:

Apples

Apples are green, or sometimes red.

Pears

Pears are green.

Oranges

Oranges are orange.

Tabs can also be grouped, so that changing the current tab in one area changes all tabs with the same name throughout the page. For example:

Linux

Linux Line 1

macOS

macOS Line 1

Windows

Windows Line 1

Linux

Linux Line 2

macOS

macOS Line 2

Windows

Windows Line 2

In this latter case, we're using `.. group-tab::` instead of simply `.. tab::`. Under the hood, we're using the sphinx-tabs extension that's included in the Zephyr setup. Within a tab, you can have most any content *other than a heading* (code-blocks, ordered and unordered lists, pictures, paragraphs, and such). You can read more about sphinx-tabs from the link above.

Instruction Steps

Also introduced in the *Getting Started Guide* is a style that makes it easy to create tutorial guides with clearly identified steps. Add the `.. rst-class:: numbered-step` directive immediately before a second-level heading (by project convention, a heading underlined with asterisks `******`, and it will be displayed as a numbered step, sequentially numbered within the document. For example:

```
.. rst-class:: numbered-step

Put your right hand in
**********************
```

Put your right hand in

See the doc/getting_started/index.rst source file and compare with the *Getting Started Guide* to see a full example. As implemented, only one set of numbered steps is intended per document.

For instructions on building the documentation, see *Documentation Generation*.

3.10 Contribution Workflow

One general practice we encourage, is to make small, controlled changes. This practice simplifies review, makes merging and rebasing easier, and keeps the change history clear and clean.

When contributing to the Zephyr Project, it is also important you provide as much information as you can about your change, update appropriate documentation, and test your changes thoroughly before submitting.

The general GitHub workflow used by Zephyr developers uses a combination of command line Git commands and browser interaction with GitHub. As it is with Git, there are multiple ways of getting a task done. We'll describe a typical workflow here:

1. Create a Fork of Zephyr to your personal account on GitHub. (Click on the fork button in the top right corner of the Zephyr project repo page in GitHub.)
2. On your development computer, change into the `zephyr` folder that was created when you *obtained the code*:

   ```
   cd zephyrproject/zephyr
   ```

   Rename the default remote pointing to the upstream repository from `origin` to `upstream`:

   ```
   git remote rename origin upstream
   ```

   Let Git know about the fork you just created, naming it `origin`:

   ```
   git remote add origin https://github.com/<your github id>/zephyr
   ```

   and verify the remote repos:

   ```
   git remote -v
   ```

   The output should look similar to:

   ```
   origin   https://github.com/<your github id>/zephyr (fetch)
   origin   https://github.com/<your github id>/zephyr (push)
   upstream https://github.com/zephyrproject-rtos/zephyr (fetch)
   upstream https://github.com/zephyrproject-rtos/zephyr (push)
   ```

3. Create a topic branch (off of main) for your work (if you're addressing an issue, we suggest including the issue number in the branch name):

   ```
   git checkout main
   git checkout -b fix_comment_typo
   ```

   Some Zephyr subsystems do development work on a separate branch from main so you may need to indicate this in your checkout:

   ```
   git checkout -b fix_out_of_date_patch origin/net
   ```

4. Make changes, test locally, change, test, test again, ... (Check out the prior chapter on *twister* as well).

5. When things look good, start the pull request process by adding your changed files:

   ```
   git add [file(s) that changed, add -p if you want to be more specific]
   ```

   You can see files that are not yet staged using:

   ```
   git status
   ```

6. Verify changes to be committed look as you expected:

   ```
   git diff --cached
   ```

7. Commit your changes to your local repo:

   ```
   git commit -s
   ```

   The `-s` option automatically adds your `Signed-off-by:` to your commit message. Your commit will be rejected without this line that indicates your agreement with the *DCO*. See the *Commit Guidelines* section for specific guidelines for writing your commit messages.

8. Push your topic branch with your changes to your fork in your personal GitHub account:

   ```
   git push origin fix_comment_typo
   ```

9. In your web browser, go to your forked repo and click on the `Compare & pull request` button for the branch you just worked on and you want to open a pull request with.

10. Review the pull request changes, and verify that you are opening a pull request for the appropriate branch. The title and message from your commit message should appear as well.

11. If you're working on a subsystem branch that's not `main`, you may need to change the intended branch for the pull request here, for example, by changing the base branch from `main` to `net`.

12. GitHub will assign one or more suggested reviewers (based on the CODEOWNERS file in the repo). If you are a project member, you can select additional reviewers now too.

13. Click on the submit button and your pull request is sent and awaits review. Email will be sent as review comments are made, or you can check on your pull request at https://github.com/zephyrproject-rtos/zephyr/pulls.

14. While you're waiting for your pull request to be accepted and merged, you can create another branch to work on another issue. (Be sure to make your new branch off of main and not the previous branch.):

    ```
    git checkout main
    git checkout -b fix_another_issue
    ```

    and use the same process described above to work on this new topic branch.

15. If reviewers do request changes to your patch, you can interactively rebase commit(s) to fix review issues. In your development repo:

    ```
    git fetch --all
    git rebase --ignore-whitespace upstream/main
    ```

    The `--ignore-whitespace` option stops `git apply` (called by rebase) from changing any whitespace. Continuing:

    ```
    git rebase -i <offending-commit-id>^
    ```

    In the interactive rebase editor, replace `pick` with `edit` to select a specific commit (if there's more than one in your pull request), or remove the line to delete a commit entirely. Then edit files to fix the issues in the review.

    As before, inspect and test your changes. When ready, continue the patch submission:

    ```
    git add [file(s)]
    git rebase --continue
    ```

    Update commit comment if needed, and continue:

    ```
    git push --force origin fix_comment_typo
    ```

    By force pushing your update, your original pull request will be updated with your changes so you won't need to resubmit the pull request.

    Note: While amending commits and force pushing is a common review model outside GitHub, and the one recommended by Zephyr, it's not the main model supported by GitHub. Forced pushes can cause unexpected behavior, such as not being able to use "View Changes" buttons except for the last one - GitHub complains it can't find older commits. You're also not always able to compare the latest reviewed version with the latest submitted version. When rewriting history GitHub only guarantees access to the latest version.

    ---

16. If the CI run fails, you will need to make changes to your code in order to fix the issues and amend your commits by rebasing as described above. Additional information about the CI system can be found in *Continuous Integration*.

3.11 Commit Guidelines

Changes are submitted as Git commits. Each commit message must contain:

- A short and descriptive subject line that is less than 72 characters, followed by a blank line. The subject line must include a prefix that identifies the subsystem being changed, followed by a colon, and a short title, for example: `doc: update wiki references to new site`. (If you're updating an existing file, you can use `git log <filename>` to see what developers used as the prefix for previous patches of this file.)
- A change description with your logic or reasoning for the changes, followed by a blank line.
- A Signed-off-by line, `Signed-off-by: <name> <email>` typically added automatically by using `git commit -s`
- If the change addresses an issue, include a line of the form:

  ```
  Fixes #<issue number>.
  ```

All changes and topics sent to GitHub must be well-formed, as described above.

3.11.1 Commit Message Body

When editing the commit message, please briefly explain what your change does and why it's needed. A change summary of `"Fixes stuff"` will be rejected.

Warning: An empty change summary body is not permitted. Even for trivial changes, please include a summary body in the commit message.

The description body of the commit message must include:

- **what** the change does,
- **why** you chose that approach,
- **what** assumptions were made, and
- **how** you know it works – for example, which tests you ran.

For examples of accepted commit messages, you can refer to the Zephyr GitHub changelog.

3.11.2 Other Commit Expectations

- Commits must build cleanly when applied on top of each other, thus avoiding breaking bisectability.
- Commits must pass all CI checks (see *Continuous Integration* for more information)
- Each commit must address a single identifiable issue and must be logically self-contained. Unrelated changes should be submitted as separate commits.
- You may submit pull request RFCs (requests for comments) to send work proposals, progress snapshots of your work, or to get early feedback on features or changes that will affect multiple areas in the code base.
- When major new functionality is added, tests for the new functionality MUST be added to the automated test suite. All new APIs MUST be documented and tested and tests MUST cover at least 80% of the added functionality using the code coverage tool and reporting provided by the project.

3.11.3 Submitting Proposals

You can request a new feature or submit a proposal by submitting an issue to our GitHub Repository. If you would like to implement a new feature, please submit an issue with a proposal (RFC) for your work first, to be sure that we can use it. Please consider what kind of change it is:

- For a Major Feature, first open an issue and outline your proposal so that it can be discussed. This will also allow us to better coordinate our efforts, prevent duplication of work, and help you to craft the change so that it is successfully accepted into the project. Providing the following information will increase the chances of your issue being dealt with quickly:
  - Overview of the Proposal
  - Motivation for or Use Case
  - Design Details
  - Alternatives
  - Test Strategy
- Small Features can be crafted and directly submitted as a Pull Request.

3.11.4 Identifying Contribution Origin

When adding a new file to the tree, it is important to detail the source of origin on the file, provide attributions, and detail the intended usage. In cases where the file is an original to Zephyr, the commit message should include the following ("Original" is the assumption if no Origin tag is present):

```
Origin: Original
```

In cases where the file is *imported from an external project*, the commit message shall contain details regarding the original project, the location of the project, the SHA-id of the origin commit for the file and the intended purpose.

For example, a copy of a locally maintained import:

```
Origin: Contiki OS
License: BSD 3-Clause
URL: http://www.contiki-os.org/
commit: 853207acfdc6549b10eb3e44504b1a75ae1ad63a
Purpose: Introduction of networking stack.
```

For example, a copy of an externally maintained import in a module repository:

```
Origin: Tiny Crypt
License: BSD 3-Clause
URL: https://github.com/01org/tinycrypt
commit: 08ded7f21529c39e5133688ffb93a9d0c94e5c6e
Purpose: Introduction of TinyCrypt
```

3.12 Continuous Integration (CI)

The Zephyr Project operates a Continuous Integration (CI) system that runs on every Pull Request (PR) in order to verify several aspects of the PR:

- Git commit formatting
- Coding Style
- Twister builds for multiple architectures and boards
- Documentation build to verify any doc changes

CI is run on Github Actions and it uses the same tools described in the *Contribution Tools* section. The CI results must be green indicating "All checks have passed" before the Pull Request can be merged. CI is run when the PR is created, and again every time the PR is modified with a commit.

The current status of the CI run can always be found at the bottom of the GitHub PR page, below the review status. Depending on the success or failure of the run you will see:

- "All checks have passed"
- "All checks have failed"

In case of failure you can click on the "Details" link presented below the failure message in order to navigate to `Github Actions` and inspect the results. Once you click on the link you will be taken to the `Github actions` summary results page where a table with all the different builds will be shown. To see what build or test failed click on the row that contains the failed (i.e. non-green) build.

The builds@lists.zephyrproject.org mailing list archives any nightly build results produced by CI.

3.13 Contributions to External Modules

Follow the guidelines in the *Modules (External projects)* section for contributing *new modules* and submitting changes to *existing modules*.

3.14 Contributing External Components

3.14.1 Contributing source code from external projects

In some cases it is desirable to leverage existing, external source code in order to avoid re-implementing basic functionality or features that are readily available in other open source projects.

This section describes the circumstances under which external source code can be imported into Zephyr, and the process that governs the inclusion.

There are three main factors that will be considered during the inclusion process in order to determine whether it will be accepted. These will be described in the following sections.

Software License

Note: External source code licensed under the Apache-2.0 license is not subject to this section.

Integrating code into the Zephyr Project from other projects that use a license other than the Apache 2.0 license needs to be fully understood in context and approved by the Zephyr governing board, as described in the Zephyr project charter. The board will automatically reject licenses that have not been approved by the Open Source Initiative (OSI). See the *Submission and review process* section for more details.

By carefully reviewing potential contributions and also enforcing a *Developer Certification of Origin (DCO)* for contributed code, we ensure that the Zephyr community can develop products with the Zephyr Project without concerns over patent or copyright issues.

Merit

Just like with any other regular contribution, one that contains external code needs to be evaluated for merit. However, in the particular case of code that comes from an existing project, there are additional questions that must be answered in order to accept the contribution. More specifically, the following will be considered by the Technical Steering Committee and evaluated carefully before the external source code is accepted into the project:

- Is this the most optimal way to introduce the functionality to the project? Both the cost of implementing this internally and the one incurred in maintaining an externally developed codebase need to be evaluated.
- Is the external project being actively maintained? This is particularly important for source code that deals with security or cryptography.
- Have alternatives to the particular implementation proposed been considered? Are there other open source project that implement the same functionality?

Mode of integration

There are two ways of integrating external source code into the Zephyr Project, and careful consideration must be taken to choose the appropriate one for each particular case.

Integration in the main tree The first way to integrate external source code into the project is to simply import the source code files into the main `zephyr` repository. This automatically implies that the imported source code becomes part of the "mainline" codebase, which in turn requires that:

- The code is formatted according to the Zephyr *Coding Style*
- The code adheres to the project's *Coding Guidelines*
- The code is subject to the same checks and verification requirements as the rest of the code in the main tree, including static analysis
- All files contain an SPDX tag if not already present
- An entry is added to the *licensing page*

This mode of integration can be applicable to both small and large external codebases, but it is typically used more commonly with the former.

Integration as a module The second way of integrating external source code into the project is to import the whole or parts of the third-party open source project into a separate repository, and then include it under the form of a *module*. With this approach the code is considered as being developed externally, and thus it is not automatically subject to the requirements of the previous section.

Ongoing maintenance

Regardless of the mode of integration, external source code that is integrated in Zephyr requires regular ongoing maintenance. The submitter of the proposal to integrate external source code must therefore commit to maintain the integration of such code for the foreseeable future. This may require adding an entry in the `MAINTAINERS.yaml` as part of the process.

Submission and review process

Before external source code can be included in the project, it must be reviewed and accepted by the Technical Steering Committee (TSC) and, in some cases, by the Zephyr governing board.

A request for external source code integration must be made by creating a new issue in the Zephyr project issue tracking system on GitHub with details about the source code and how it integrates into the project.

Follow the steps below to begin the submission process:

1. Make sure to read through the *Contributing source code from external projects* section in detail, so that you are informed of the criteria used by the TSC and board in order to approve or reject a request
2. Use the New External Source Code Issue to open an issue
3. Fill out all required sections, making sure you provide enough detail for the TSC to assess the merit of the request. Optionally you can also create a Pull Request that demonstrates the integration of the external source code and link to it from the issue
4. Wait for feedback from the TSC, respond to any additional questions added as GitHub issue comments

If, after consideration by the TSC, the conclusion is that integrating external source code is the best solution, and the external source code is licensed under the Apache-2.0 license, the submission process is complete and the external source code can be integrated.

If, however, the external source code uses a license other than Apache-2.0, then these additional steps must be followed:

1. The TSC chair will forward the link to the GitHub issue created during the early submission process to the Zephyr governing board for further review

2. The Zephyr governing board has two weeks to review and ask questions:
   - If there are no objections, the matter is closed. Approval can be accelerated by unanimous approval of the board before the two weeks are up
   - If a governing board member raises an objection that cannot be resolved via email, the board will meet to discuss whether to override the TSC approval or identify other approaches that can resolve the objections
3. On approval of the Zephyr TSC and governing board the submission process is complete

The flowchart below shows an overview of the process:

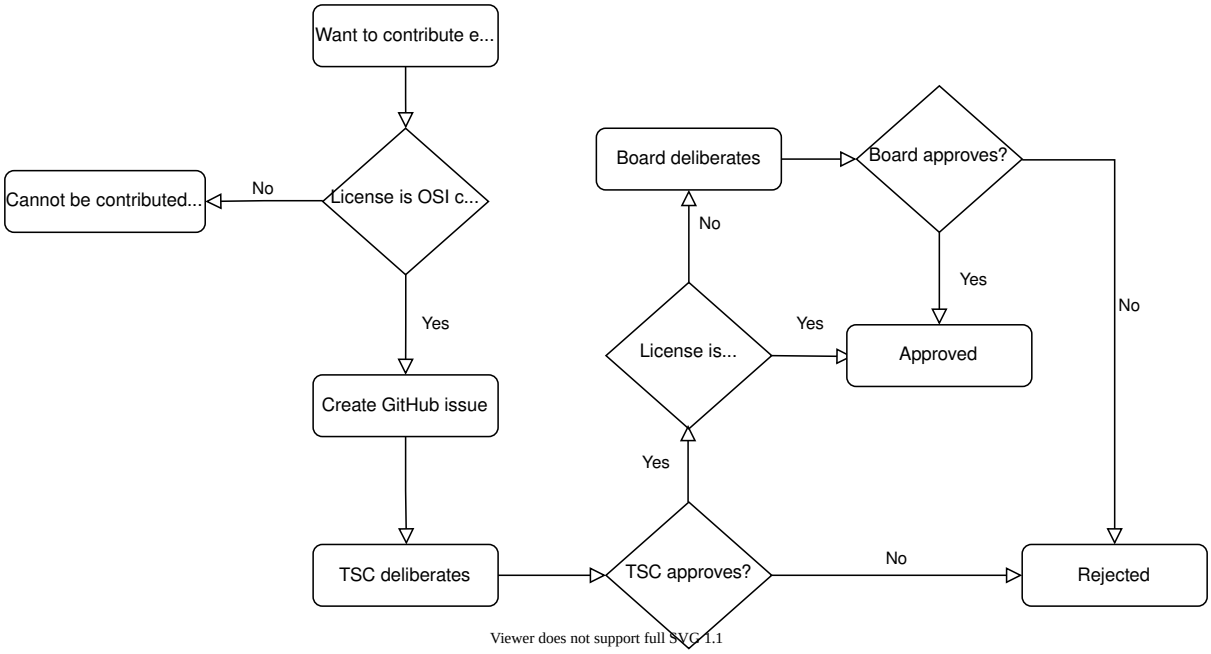


Fig. 1: Submission process

Chapter 4

Development and Contribution Process

4.1 TSC Project Roles

4.1.1 Main Roles

TSC projects generally will involve *Maintainers*, *Collaborators*, and *Contributors*:

Maintainer: lead Collaborators on an area identified by the TSC (e.g. Architecture, code subsystems, etc.). Maintainers shall also serve as the area's representative on the TSC as needed. Maintainers may become voting members of the TSC under the guidelines stated in the project Charter.

Collaborator: A highly involved Contributor in one or more areas. May become a Maintainer with approval of existing TSC voting members.

Contributor: anyone in the community that contributes code or documentation to the project. Contributors may become Collaborators by approval of the existing Collaborators and Maintainers of the particular code base areas or subsystems.

Contributor

A *Contributor* is a developer who wishes to contribute to the project, at any level. Contributors who show dedication and skill are rewarded with additional rights and responsibilities.

Contributors are granted the following rights and responsibilities:

- Right to contribute code, documentation, translations, artwork, etc.
- Right to report defects (bugs) and suggestions for enhancement.
- Right to participate in the process of reviewing contributions by others.
- Right to initiate and participate in discussions in any communication methods.
- Right to approach any member of the community with matters they believe to be important.
- Right to participate in the feature development process.
- Responsibility to abide by decisions, once made. They are welcome to provide new, relevant information to reopen decisions.
- Responsibility for issues and bugs introduced by one's own contributions.
- Responsibility to respect the rules of the community.
- Responsibility to provide constructive advice whenever participating in discussions and in the review of contributions.

- Responsibility to follow the project's code of conduct (https://github.com/zephyrproject-rtos/zephyr/blob/main/CODE_OF_CONDUCT.md)

Collaborator

A *Collaborator* is a Contributor who is also responsible for the maintenance of Zephyr source code. Their opinions weigh more when decisions are made, in a fully meritocratic fashion.

Collaborators have the following rights and responsibilities, in addition to those listed for Contributors:

- Right to set goals for the short and medium terms for the project being maintained, alongside the Maintainer.
- Responsibility to participate in the feature development process.
- Responsibility to review relevant code changes within reasonable time.
- Responsibility to ensure the quality of the code to expected levels.
- Responsibility to participate in community discussions.
- Responsibility to mentor new contributors when appropriate
- Responsibility to participate in the quality verification and release process, when those happen.

Maintainer

A *Maintainer* is a Collaborator who is also responsible for knowing, directing and anticipating the needs of a given zephyr source code area.

Maintainers have the following rights and responsibilities, in addition to those listed for Contributors and Collaborators:

- Right to set the overall architecture of the relevant subsystems or areas of involvement.
- Right to make decisions in the relevant subsystems or areas of involvement, in conjunction with the collaborators.
- Responsibility to convey the direction of the relevant subsystem or areas to the TSC
- Responsibility to ensure all contributions of the project have been reviewed within reasonable time.
- Responsibility to enforce the code of conduct.

4.1.2 Role Retirement

- Individuals elected to the following Project roles, including, Maintainer, Release Engineering Team member, Release Manager, but are no longer engaged in the project as described by the rights and responsibilities of that role, may be requested by the TSC to retire from the role they are elected.
- Such a request needs to be raised as a motion in the TSC and be approved by the TSC voting members. By approval of the TSC the individual is considered to be retired from the role they have been elected.
- The above applies to elected TSC Project roles that may be defined in addition.

4.1.3 Teams and Supporting Activities

Assignee

An *Assignee* is one of the maintainers of a subsystem or code being changed. Assignees are set either automatically based on the code being changed or set by the other Maintainers, the Release Engineering team can set an assignee when the latter is not possible.

- Right to dismiss stale reviews and seek reviews from additional maintainers, developers and contributors
- Right to block pull requests from being merged
- Responsibility to re-assign a pull request if they are the original submitter of the code
- Responsibility to drive the pull request to a mergeable state
- Solicit approvals from maintainers of the subsystems affected
- Responsibility to drive the escalation process

Release Engineering Team

A team of active Maintainers involved in multiple areas.

- The members of the Release Engineering team are expected to fill the Release Manager role based on a defined cadence and selection process.
- The cadence and selection process are defined by the Release Engineering team and are approved by the TSC.
- The team reports directly into the TSC.

Release Engineering team has the following rights and responsibilities:

- Right to merge code changes to the zephyr tree following the project rules.
- Right to revert any changes that have broken the code base
- Right to close any stale changes after <N> months of no activity
- Responsibility to take directions from the TSC and follow them.
- Responsibility to coordinate code merges with maintainers.
- Responsibility to merge all contributions regardless of their origin and area if they have been approved by the respective maintainers and follow the merge criteria of a change.
- Responsibility to keep the Zephyr code base in a working and passing state (as per CI)

Joining the Release Engineering team

- Maintainers highly involved in the project may be nominated by a TSC voting member to join the Release Engineering team. Nominees may become members of the team by approval of the existing TSC voting members.
- To ensure a functional Release Engineering team the TSC shall periodically review the team's followed processes, the appropriate size, and the membership composition (ensure, for example, that team members are geographically distributed across multiple locations and time-zones).

Release Manager

A *Maintainer* responsible for driving a specific release to completion following the milestones and the roadmap of the project for this specific release.

- TSC has to approve a release manager.

A Release Manager is a member of the Release Engineering team and has the rights and responsibilities of that team in addition to the following:

- Right to manage and coordinate all code merges after the code freeze milestone (M3, see program management overview.)
- Responsibility to drive and coordinate the triaging process for the release
- Responsibility to create the release notes of the release
- Responsibility to notify all stakeholders of the project, including the community at large about the status of the release in a timely manner.
- Responsibility to coordinate with QA and validation and verify changes either directly or through QA before major changes and major milestones.

Roles / Permissions

Table 1: Project Roles vs Github Permissions

		Admin	**Merge Rights**	Member	Owner	Collaborator
Main Roles	Contributor					x
	Collaborator			x		
	Maintainer			x		
Supportive Roles	QA/Validation			x		x
	DevOps	**x**				
	System Admin	**x**			x	
	Release Engineering	**x**	**x**	x		

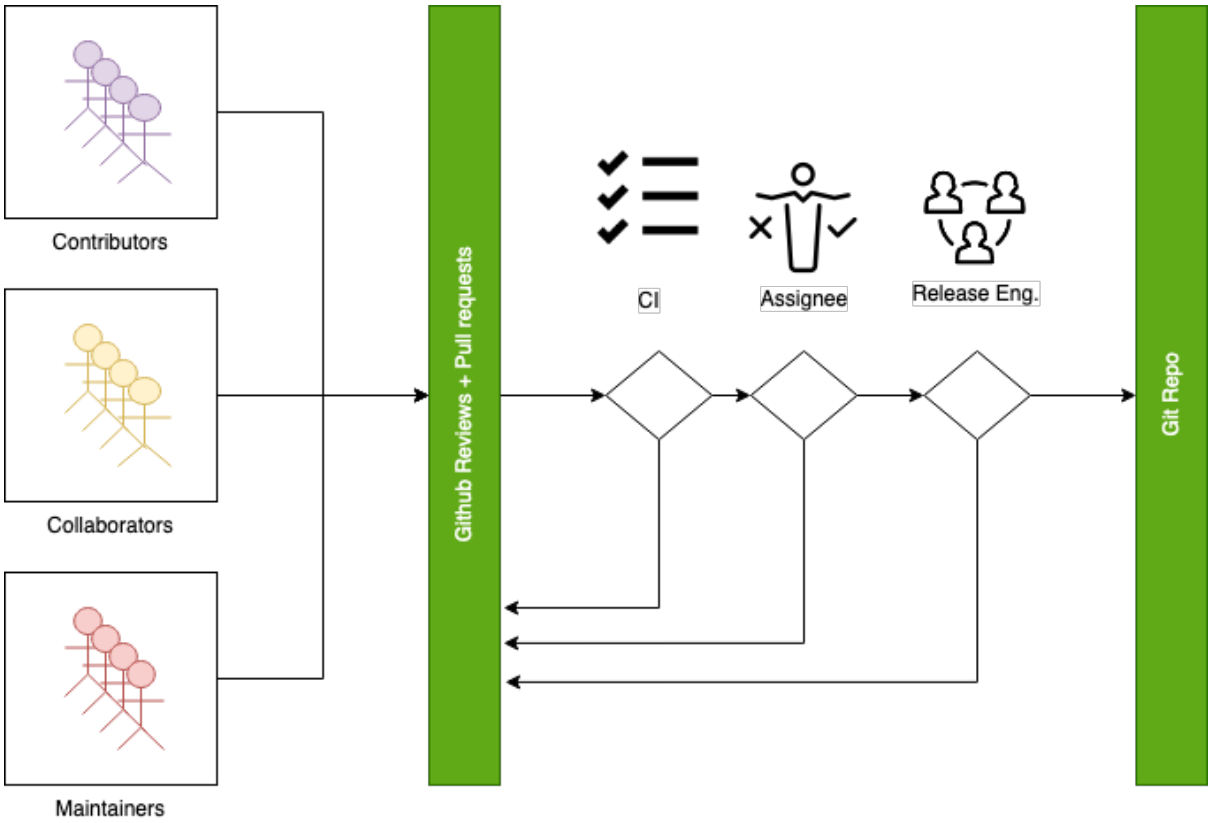
4.1.4 MAINTAINERS File

Generic guidelines for deciding and filling in the Maintainers' list

- The MAINTAINERS file shall replace the CODEOWNERS file and will be used for both setting assignees and reviewers.
- We should keep the granularity of code maintainership at a manageable level
- We should be looking for maintainers for areas of code that are orphaned (i.e. without an explicit maintainer)
  - Un-maintained areas should be indicated clearly in the MAINTAINERS file
- All submitted pull-requests should have an assignee
- We Introduce an area/subsystem hierarchy to address the above point
  - Parent-area maintainer should be acting as default substitute/fallback assignee for un-maintained sub-areas
  - Area maintainer gets precedence over parent-area maintainer
- Pull-requests may be re-assigned if this is needed or more appropriate
  - Re-assigned by original assignee (see "Assignee" slide)
- In general, updates to the MAINTAINERS file should be in a standalone commit alongside other changes introducing new files and directories to the tree.
- Major changes to the file, including the addition of new areas with new maintainers should come in as standalone pull-requests and require TSC review.
- If additional review by the TSC is required, the maintainers of the file should send the requested changes to the TSC and give members of the TSC two (2) days to object to any of the changes to maintainership of areas or the addition of new maintainers or areas.
- Path, collaborator and name changes do not require a review by the TSC.

- Addition of new areas without a maintainer do not require review by the TSC.
- The MAINTAINERS file itself shall have a maintainer
- Architectures, core components, sub-systems, samples, tests
  - Each area shall have an explicit maintainer
- Boards (incl relevant samples, tests), SoCs (incl DTS) * May have a maintainer, shall have a higher-level platform maintainer
- Drivers
  - Shall have a driver-area (and API) maintainer
  - Could have individual driver implementation maintainers but preferably collaborator/contributors
  - In the above case, platform-specific PRs may be re-assigned to respective collaborator/contributor of driver implementation

4.1.5 Release Activity



Merge Criteria

- All continuous integration checks have passed
  - Codeowners
  - Device Tree
  - Documentation

  - Gitlint
  - Identity/Emails
  - Kconfig
  - License
  - Checkpatch (Coding Style)
  - Pylint
  - Integration Tests (Via twister) on emulation/simulation platforms
  - Simulated Bluetooth Tests
- Planned
  - Footprint
  - Code coverage
  - Coding Guidelines
  - Static Analysis (Coverity)
  - Documentation coverage (APIs)
- PR template with checklist
- Minimal of 2 approvals
  - A collaborator from the same subsystem.
  - Alternately another maintainer of another subsystem
  - Approval by the assignee
- A minimum review period of 2 days, 4 hours for trivial changes (see *Give reviewers time to review before code merge*). Hotfixes can be merged at any time after CI passes.
- All required checks are passing

Escalation Process

- Contributors may object to change requests or decisions made by Maintainers.
- Process
  - Resolve in the PR among assignee, maintainers and reviewer
    - Assignee to act as moderator if applicable
  - Optionally resolve in the dev review meeting with more Maintainers and project stakeholders
    - The involved parties and the Assignee to be present when the (escalated) issue is discussed
  - TSC: Assignees can escalate to the TSC voting members and get a binding resolution in the TSC.
  - Assignee to ensure the resolution of the escalation is reflected in the PR review.

4.2 Release Process

The Zephyr project releases on a time-based cycle, rather than a feature-driven one. Zephyr releases represent an aggregation of the work of many contributors, companies, and individuals from the community.

A time-based release process enables the Zephyr project to provide users with a balance of the latest technologies and features and excellent overall quality. A roughly 4-month release cycle allows the project to coordinate development of the features that have actually been implemented, allowing the project to maintain the quality of the overall release without delays because of one or two features that are not ready yet.

The Zephyr release model is loosely based on the Linux kernel model:

- Release tagging procedure:
  - linear mode on main branch,
  - release branches for maintenance after release tagging.
- Each release period will consist of a merge window period followed by one or more release candidates on which only stabilization changes, bug fixes, and documentation can be merged in.
  - Merge window mode: all changes are accepted (subject to approval from the respective maintainers.)
  - When the merge window is closed, the release owner lays a vN-rc1 tag and the tree enters the release candidate phase
  - CI sees the tag, builds and runs tests; QA analyses the report from the build and test run and gives an ACK/NAK to the build
  - The release owner, with QA and any other needed input, determines if the release candidate is a go for release
  - If it is a go for a release, the release owner lays a tag release vN at the same point
- Development on new features continues in topic branches. Once features are ready, they are submitted to mainline during the merge window period and after the release is tagged.

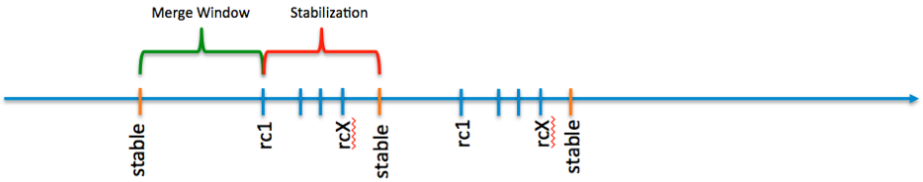


Fig. 1: Release Cycle

4.2.1 Merge Window

A relatively straightforward discipline is followed with regard to the merging of patches for each release. At the beginning of each development cycle, the “merge window” is said to be open. At that time, code which is deemed to be sufficiently stable (and which is accepted by the development community) is merged into the mainline tree. The bulk of changes for a new development cycle (and all of the major changes) will be merged during this time.

The merge window lasts for approximately two months. At the end of this time, the release owner will declare that the window is closed and release the first of the release candidates. For the codebase release which is destined to be 0.4.0, for example, the release which happens at the end of the merge window will be called 0.4.0-rc1. The -rc1 release is the signal that the time to merge new features has passed, and that the time to stabilize the next release of the code base has begun.

Over the next weeks, only patches which fix problems should be submitted to the mainline. On occasion, a more significant change will be allowed, but such occasions are rare and require a TSC approval (Change Control Board). As a general rule, if you miss the merge window for a given feature, the best thing to do is to wait for the next development cycle. (An occasional exception is made for drivers

for previously unsupported hardware; if they do not touch any other in-tree code, they cannot cause regressions and should be safe to add at any time).

As fixes make their way into the mainline, the patch rate will slow over time. The mainline release owner releases new -rc drops once or twice a week; a normal series will get up to somewhere between -rc4 and -rc6 before the code base is considered to be sufficiently stable and the quality metrics have been achieved at which point the final 0.4.x release is made.

At that point, the whole process starts over again.

Here is the description of the various moderation levels:

- Low:
  - Major New Features
  - Bug Fixes
  - Refactoring
  - Structure/Directory Changes
- Medium:
  - Bug Fixes, all priorities
  - Enhancements
  - Minor "self-contained" New Features
- High:
  - Bug Fixes: P1 and P2
  - Documentation + Test Coverage

4.2.2 Release Quality Criteria

The current backlog of prioritized bugs shall be used as a quality metric to gate the final release. The following counts shall be used:

Table 2: Bug Count Release Thresholds

High	Medium	Low
0	<20	<50

Note: The "low" bug count target of <50 will be a phased appoach starting with 150 for release 2.4.0, 100 for release 2.5.0, and 50 for release 2.6.0

4.2.3 Releases

The following syntax should be used for releases and tags in Git:

- Release [Major].[Minor].[Patch Level]
- Release Candidate [Major].[Minor].[Patch Level]-rc[RC Number]
- Tagging:
  - v[Major].[Minor].[Patch Level]-rc[RC Number]
  - v[Major].[Minor].[Patch Level]

– v[Major].[Minor].99 - A tag applied to main branch to signify that work on v[Major].[Minor+1] has started. For example, v1.7.99 will be tagged at the start of v1.8 process. The tag corresponds to VERSION_MAJOR/VERSION_MINOR/PATCHLEVEL macros as defined for a work-in-progress main branch version. Presence of this tag allows generation of sensible output for "git describe" on main branch, as typically used for automated builds and CI tools.

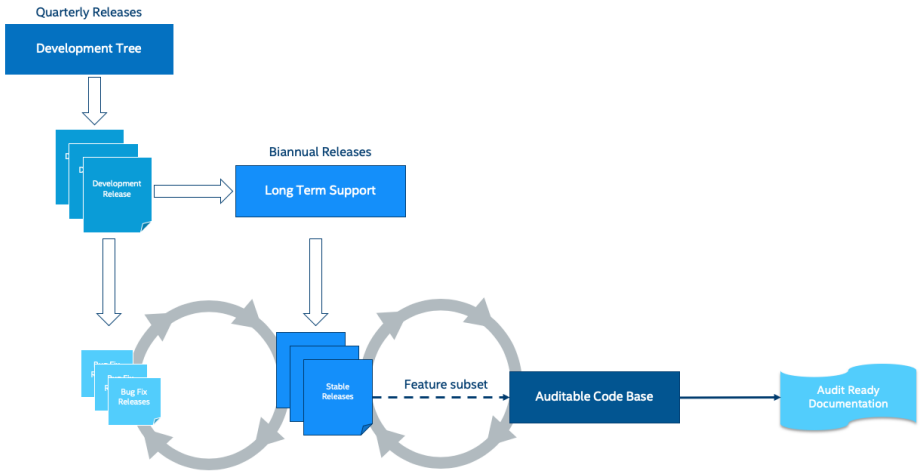


Fig. 2: Zephyr Code and Releases

Long Term Support (LTS)

Long-term support releases are designed to be supported and maintained for an extended period and is the recommended release for products and the auditable branch used for certification.

An LTS release is defined as:

- **Product focused**
- **Extended Stabilisation period**: Allow for more testing and bug fixing
- **Stable APIs**
- **Quality Driven Process**
- **Long Term**: Maintained for an extended period of time (at least 2.5 years) overlapping previous LTS release for at least half a year.

Product Focused Zephyr LTS is the recommended release for product makers with an extended support and maintenance which includes general stability and bug fixes, security fixes.

An LTS includes both mature and new features. API and feature maturity is documented and tracked. The footprint and scope of mature and stable APIs expands as we move from one LTS to the next giving users access to bleading edge features and new hardware while keeping a stable foundation that evolves over time.

Extended Stabilisation Period Zephyr LTS development cycle differs from regular releases and has an extended stabilization period. Feature freeze of regular releases happens 3-4 weeks before the scheduled release date. The stabilisation period for LTS is extended by 3 weeks with the feature freeze occurring 6-7 weeks before the anticipated release date. The time between code freeze and release date is extended in this case.

Stable APIs Zephyr LTS provides a stable and long-lived foundation for developing products. To guarantee stability of the APIs and the implementation of such APIs it is required that any release software that makes the core of the OS went through the Zephyr API lifecycle and stabilised over at least 2 releases. This guarantees that we release many of the highlighted and core features with mature and well-established implementations with stable APIs that are supported during the lifetime of the release LTS.

- API Freeze (LTS - 2)
  - All stable APIs need to be frozen 2 releases before an LTS. APIs can be extended with additional features, but the core implementation is not modified. This is valid for the following subsystems for example:
    - Device Drivers (i2c.h, spi.h)...
    - Kernel (k_*):
    - OS services (logging,debugging, ..)
    - DTS: API and bindings stability
    - Kconfig
  - New APIs for experimental features can be added at any time as long as they are standalone and documented as experimental or unstable features/APIs.
- Feature Freeze (LTS - 1) - No new features or overhaul/restructuring of code covering major LTS features.
  - Kernel + Base OS
  - Additional advertised LTS features
  - Auxiliary features on top of and/or extending the base OS and advertised LTS features can be added at any time and should be marked as experimental if applicable

Quality Driven Process The Zephyr project follows industry standards and processes with the goal of providing a quality oriented releases. This is achieved by providing the following products to track progress, integrity and quality of the software components provided by the project:

- Compliance with pubished coding guidelines, style guides and naming conventions and documentation of deviations.
- Regular static analysis on the complete tree using available commercial and open-source tools and documentation of deviations and false positives.
- Documented components and APIS
- Requirements Catalog
- Verification Plans
- Verification Reports
- Coverage Reports
- Requirements Traceability Matrix (RTM)
- SPDX License Reports

Each release is created with the above products to document the quality and the state of the software when it was released.

Long Term Support and Maintenance A Zephyr LTS release is published every 2 years and is branched and maintained independently from the main tree for at least 2.5 years after it was released. Support and maintenance for an LTS release stops at least half a year after the following LTS release is published.

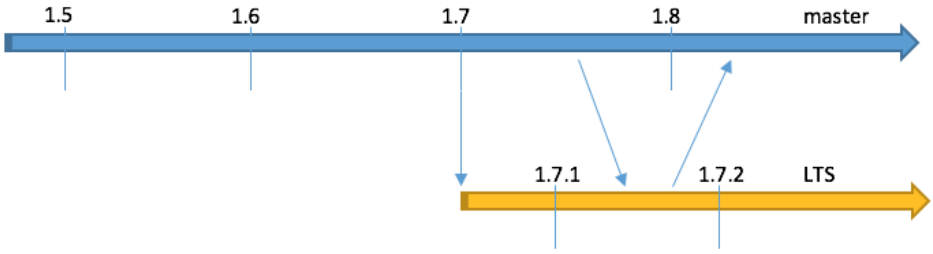


Fig. 3: Long Term Support Release

Changes and fixes flow in both directions. However, changes from main branch to an LTS branch will be limited to fixes that apply to both branches and for existing features only.

All fixes for an LTS branch that apply to the mainline tree shall be submitted to mainline tree as well.

Auditable Code Base

An auditable code base is to be established from a defined subset of Zephyr OS features and will be limited in scope. The LTS, development tree, and the auditable code bases shall be kept in sync after the audit branch is created, but with a more rigorous process in place for adding new features into the audit branch used for certification.

This process will be applied before new features move into the auditable code base.

The initial and subsequent certification targets will be decided by the Zephyr project governing board.

Processes to achieve selected certification will be determined by the Security and Safety Working Groups and coordinated with the TSC.

4.2.4 Release Procedure

This section documents the Release manager responsibilities so that it serves as a knowledge repository for Release managers.

Milestones

The following graphic shows the timeline of phases and milestones associated with each release:

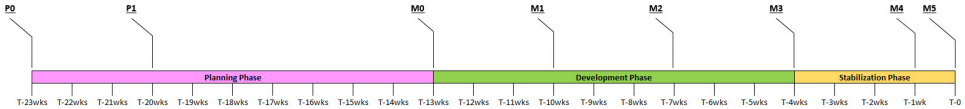


Fig. 4: Release milestones

This shows how the phases and milestones of one release overlap with those of the next release:

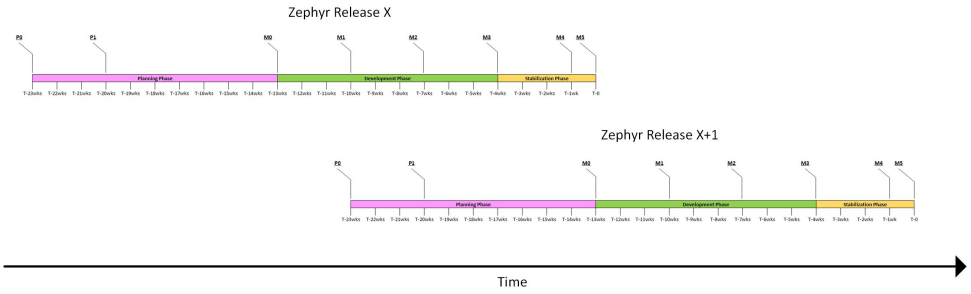


Fig. 5: Release milestones with planning

Table 3: Milestone Description

Mile-stone	Descrip-tion	Definition
P0	Planning Kickoff	Start Entering Requirements
P1		TSC Agrees on Major Features and Schedule
M0	Merge Window Open	All features, Sized, and AssignedMerge Window Is Opened
M1	M1 Check-point	Major Features Ready for Code Reviews Test Plans Reviewed and Approved
M2	Feature Merge Window Close	Feature Freeze Feature Development Complete (including Code Reviews and Unit Tests Passing) P1 Stories Implemented Feature Merge Window Is Closed Test Development Complete Technical Documentation Created/Updated and Ready for Review CCB Control Starts
M3	Code Freeze	Code Freeze RC3 Tagged and Built
M4	Release	TSC Reviews the Release Criteria Report and Approves Release Final RC Tagged Make the Release

Release Checklist

Each release has a GitHub issue associated with it that contains the full checklist. After a release is complete, a checklist for the next release is created.

Tagging

The final release and each release candidate shall be tagged using the following steps:

Note: Tagging needs to be done via explicit git commands and not via GitHub's release interface. The GitHub release interface does not generate annotated tags (it generates 'lightweight' tags regardless of release or pre-release). You should also upload your gpg public key to your GitHub account, since the instructions below involve creating signed tags. However, if you do not have a gpg public key you can opt to remove the `-s` option from the commands below.

Release Candidate

Note: This section uses tagging 1.11.0-rc1 as an example, replace with the appropriate release candidate

version.

1. Update the version variables in the VERSION file located in the root of the Git repository to match the version for this release candidate. The `EXTRAVERSION` variable is used to identify the rc[RC Number] value for this candidate:

   ```
   EXTRAVERSION = rc1
   ```

2. Post a PR with the updated VERSION file using `release: Zephyr 1.11.0-rc1` as the commit subject. Merge the PR after successful CI.

3. Tag and push the version, using an annotated tag:

   ```
   $ git pull
   $ git tag -s -m "Zephyr 1.11.0-rc1" v1.11.0-rc1
   $ git push git@github.com:zephyrproject-rtos/zephyr.git v1.11.0-rc1
   ```

4. Once the tag is pushed, a github action will create a draft release in Github with a shortlog since the last tag. The action will also create a SPDX manifest of the Zephyr tree and will add the file as an asset in the release.

   Go to the draft release that was created and edit as needed. If this step fails for a reason, it can be done manually following the steps below:

   1. Create a shortlog of changes between the previous release (use rc1..rc2 between release candidates):

      ```
      $ git shortlog v1.10.0..v1.11.0-rc1
      ```

   2. Find the new tag at the top of the releases page and edit the release with the `Edit tag` button with the following:

      - Name it `Zephyr 1.11.0-rc1`
      - Copy the shortlog into the release notes textbox (*don't forget to quote it properly so it shows as unformatted text in Markdown*)
      - Check the "This is a pre-release" checkbox

5. Send an email to the mailing lists (`announce` and `devel`) with a link to the release

Final Release

Note: This section uses tagging 1.11.0 as an example, replace with the appropriate final release version.

When all final release criteria has been met and the final release notes have been approved and merged into the repository, the final release version will be set and repository tagged using the following procedure:

1. Update the version variables in the VERSION file located in the root of the Git repository. Set `EXTRAVERSION` variable to an empty string to indicate final release:

   ```
   EXTRAVERSION =
   ```

2. Post a PR with the updated VERSION file using `release: Zephyr 1.11.0` as the commit subject. Merge the PR after successful CI.

3. Tag and push the version, using two annotated tags:

   ```
   $ git pull
   $ git tag -s -m "Zephyr 1.11.0" v1.11.0
   $ git push git@github.com:zephyrproject-rtos/zephyr.git v1.11.0
   ```

(continues on next page)

(continued from previous page)

```
# This is the tag that will represent the release on GitHub, so that
# the file you can download is named ``zephyr-v1.11.0.zip`` and not
# just ``v1.11.0.zip``
$ git tag -s -m "Zephyr 1.11.0" zephyr-v1.11.0
$ git push git@github.com:zephyrproject-rtos/zephyr.git zephyr-v1.11.0
```

4. Find the new `zephyr-v1.11.0` tag at the top of the releases page and edit the release with the `Edit tag` button with the following:
   - Name it `Zephyr 1.11.0`
   - Copy the full content of `docs/releases/release-notes-1.11.rst` into the release notes textbox
5. Send an email to the mailing lists (`announce` and `devel`) with a link to the release

Listing all closed GitHub issues

The release notes for a final release contain the list of GitHub issues that have been closed during the development process of that release.

In order to obtain the list of issues closed during the release development cycle you can do the following:

1. Look for the last release before the current one and find the day it was tagged:

```
$ git show -s --format=%ci zephyr-v1.10.0
tag zephyr-v1.10.0
Tagger: Kumar Gala <kumar.gala@linaro.org>

Zephyr 1.10.0
2017-12-08 13:32:22 -0600
```

2. Use available release tools to list all the issues that have been closed between that date and the day of the release.

4.3 Feature Tracking

For feature tracking we use Github labels to classify new features and enhancements. The following is the description of each category:

Enhancement Changes to existing features that are not considered a bug and would not block a release. This is an incremental enhancement to a feature that already exists in Zephyr.

Feature request A request for the implementation or inclusion of a new unit of functionality that is not part of any release plans yet, that has not been vetted, and needs further discussion and details.

Feature A committed and planned unit of functionality with a detailed design and implementation proposal and an owner. Features must go through an RFC process and must be vetted and discussed in the TSC before a target milestone is set.

Hardware Support A request or plan to port an existing feature or enhancement to a particular hardware platform. This ranges from porting Zephyr itself to a new architecture, SoC or board to adding an implementation of a peripheral driver API for an existing hardware platform.

Meta A label to group other GitHub issues that are part of a single feature or unit of work.

The following workflow should be used to process features:.

This is the formal way for asking for a new feature in Zephyr and indicating its importance to the project. Often, the requester may have a readiness and willingness to drive implementation of the feature in an upcoming release, and should assign the request to themselves. If not though, an owner will be assigned after evaluation by the TSC. A feature request can also have a companion RFC with more details on the feature and a proposed design or implementation.

- Label new features requests as `feature-request`
- The TSC discusses new `feature-request` items regularly and triages them. Items are examined for similarity with existing features, how they fit with the project goals and other timeline considerations. The priority is determined as follows:
  - High = Next milestone
  - Medium = As soon as possible
  - Low = Best effort
- After the initial discussion and triaging, the label is moved from `feature-request` to `feature` with the target milestone and an assignee.

All items marked as `feature-request` are non-binding and those without an assignee are open for grabs, meaning that they can be picked up and implemented by any project member or the community. You should contact an assigned owner if you'd like to discuss or contribute to that feature's implementation

4.3.1 Proposals and RFCs

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Many changes however are "substantial" and need to go through a design process and produce a consensus among the project stakeholders.

The "RFC" (request for comments) process is intended to provide a consistent and controlled path for new features to enter the project.

Contributors and project stakeholders should consider using this process if they intend to make "substantial" changes to Zephyr or its documentation. Some examples that would benefit from an RFC are:

- A new feature that creates new API surface area, and would require a feature flag if introduced.
- The modification of an existing stable API
- The removal of features that already shipped as part of Zephyr.
- The introduction of new idiomatic usage or conventions, even if they do not include code changes to Zephyr itself.

The RFC process is a great opportunity to get more eyeballs on proposals coming from contributors before it becomes a part of Zephyr. Quite often, even proposals that seem "obvious" can be significantly improved once a wider group of interested people have a chance to weigh in.

The RFC process can also be helpful to encourage discussions about a proposed feature as it is being designed, and incorporate important constraints into the design while it's easier to change, before the design has been fully implemented.

Some changes do not require an RFC:

- Rephrasing, reorganizing or refactoring
- Addition or removal of warnings
- Addition of new boards, SoCs or drivers to existing subsystems
- ...

The process in itself consists in creating a GitHub issue with the *RFC label* that documents the proposal thoroughly. There is an RFC template included in the main Zephyr GitHub repository that serves as a guideline to write a new RFC.

As with Pull Requests, RFCs might require discussion in the context of one of the Zephyr meetings in order to move it forward in cases where there is either disagreement or not enough voiced opinions in order to proceed. Make sure to either label it appropriately or include it in the corresponding GitHub project in order for it to be examined during the next meeting.

4.3.2 Roadmap and Release Plans

Project roadmaps and release plans are both important tools for the project, but they have very different purposes and should not be confused. A project roadmap communicates the high-level overview of a project's strategy, while a release plan is a tactical document designed to capture and track the features planned for upcoming releases.

- The project roadmap communicates the why; a release plan details the what
- A release plan spans only a few months; a product roadmap might cover a year or more

Project Roadmap

The project roadmap should serve as a high-level, visual summary of the project's strategic objectives and expectations.

If built properly, the roadmap can be a valuable tool for several reasons. It can help the project present its plan in a compelling way to existing and new stakeholders, to help recruit new members and it can be a helpful resource the team and community can refer to throughout the project's development, to ensure they are still executing according to plan.

As such, the roadmap should contain only strategic-level details, major project themes, epics, and goals.

Release Plans

The release plan comes into play when the project roadmap's high-level strategy is translated into an actionable plan built on specific features, enhancements, and fixes that need to go into a specific release or milestone.

The release plan communicates those features and enhancements slated for your project' next release (or the next few releases). So it acts as more of a project plan, breaking the big ideas down into smaller projects the community and main stakeholders of the project can make progress on.

Items labeled as `features` are short or long term release items that shall have an assignee and a milestone set.

4.4 Code Flow and Branches

4.4.1 Introduction

The zephyr Git repository has three types of branches:

main Which contains the latest state of development

topic-* Topic branches that are used for shared development of a new feature

vx.y-branch Branches which track maintenance releases based on a major release

Development in topic branches before features go to mainline allows teams to work independently on a subsystem or a feature, improves efficiency and turnaround time, and encourages collaboration and streamlines communication between developers.

Changes submitted to a development topic branch can evolve and improve incrementally in a branch, before they are submitted to the mainline tree for final integration.

By dedicating an isolated branch to complex features, it's possible to initiate in-depth discussions around new additions before integrating them into the official project.

4.4.2 Roles and Responsibilities

Development topic branch owners have the following responsibilities:

- Use the infrastructure and tools provided by the project (GitHub, Git)
- Review changes coming from team members and request review from branch owners when submitting changes.
- Keep the branch in sync with upstream and update on a regular basis.
- Push changes frequently to upstream using the following methods:
  - GitHub pull requests: for example, when reviews have not been done in the local branch (one-man branch).
  - Merge requests: When a set of changes has been done in a local branch and has been reviewed and tested in a topic branch.

4.5 Modifying Contributions made by other developers

4.5.1 Scenarios

Zephyr contributors and collaborators are encouraged to assist as reviewers in pull requests, so that patches may be approved and merged to Zephyr's main branch as part of the original pull requests. The authors of the pull requests are responsible for amending their original commits following the review process.

There are occasions, however, when a contributor might need to modify patches included in pull requests that are submitted by other Zephyr contributors. For instance, this is the case when:

- a developer cherry-picks commits submitted by other contributors into their own pull requests in order to:
  - integrate useful content which is part of a stale pull request, or
  - get content merged to the project's main branch as part of a larger patch
- a developer pushes to a branch or pull request opened by another contributor in order to:
  - assist in updating pull requests in order to get the patches merged to the project's main branch
  - drive stale pull requests to completion so they can be merged

4.5.2 Accepted policies

A developer who intends to cherry-pick and potentially modify patches sent by another contributor shall:

- clarify in their pull request the reason for cherry-picking the patches, instead of assisting in getting the patches merged in their original pull request, and
- invite the original author of the patches to their pull request review.

A developer who intends to force-push to a branch or pull request of another Zephyr contributor shall clarify in the pull request the reason for pushing and for modifying the existing patches (e.g. stating that it is done to drive the pull request review to completion, when the pull request author is not able to do so).

Note: Developers should try to limit the above practice to pull requests identified as *stale*. Read about how to identify pull requests as stale in *development processes and tools*

If the original patches are substantially modified, the developer can either:

- (preferably) reach out to the original author and request them to acknowledge that the modified patches may be merged while having the original sign-off line and author identity, or
- submit the modified patches as their *own* work (i.e. with their *own* sign-off line and author identity). In this case, the developer shall identify in the commit message(s) the original source the submitted work is based on (mentioning, for example, the original PR number).

Note: Contributors should uncheck the box *"Allow Edits By Maintainers"* to indicate that they do not wish their patches to be amended, inside their original branch or pull request, by other Zephyr developers.

4.6 Development Environment and Tools

4.6.1 Code Review

GitHub is intended to provide a framework for reviewing every commit before it is accepted into the code base. Changes, in the form of Pull Requests (PR) are uploaded to GitHub but don't actually become a part of the project until they've been reviewed, passed a series of checks (CI), and are approved by maintainers. GitHub is used to support the standard open source practice of submitting patches, which are then reviewed by the project members before being applied to the code base.

Pull requests should be appropriately *labeled*, and linked to any relevant *bug or feature tracking issues* .

The Zephyr project uses GitHub for code reviews and Git tree management. When submitting a change or an enhancement to any Zephyr component, a developer should use GitHub. GitHub automatically assigns a responsible reviewer on a component basis, as defined in the CODEOWNERS file stored with the code tree in the Zephyr project repository. A limited set of release managers are allowed to merge a pull request into the main branch once reviews are complete.

Give reviewers time to review before code merge

The Zephyr project is a global project that is not tied to a certain geography or timezone. We have developers and contributors from across the globe. When changes are proposed using pull request, we need to allow for a minimal review time to give developers and contributors the opportunity to review and comment on changes. There are different categories of changes and we know that some changes do require reviews by subject matter experts and owners of the subsystem being changed. Many changes fall under the "trivial" category that can be addressed with general reviews and do not need to be queued for a maintainer or code-owner review. Additionally, some changes might require further discussions and a decision by the TSC or the Security working group. To summarize the above, the diagram below proposes minimal review times for each category:

Workflow

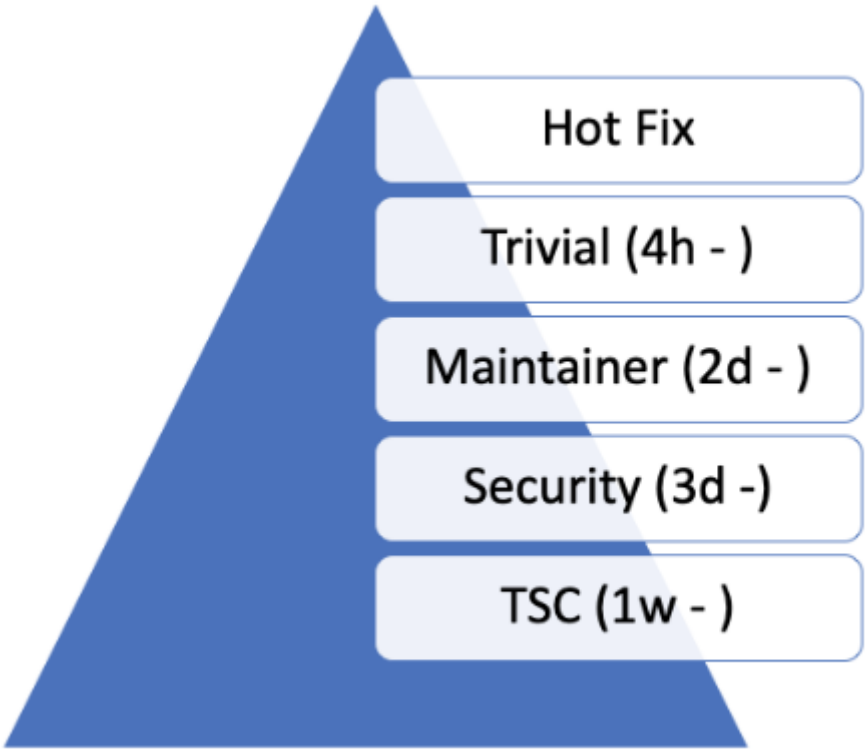


Fig. 6: Pull request classes

- An author of a change can suggest in his pull-request which category a change should belong to. A project maintainers or TSC member monitoring the inflow of changes can change the label of a pull request by adding a comment justifying why a change should belong to another category.
- The project will use the label system to categorize the pull requests.
- Changes should not be merged before the minimal time has expired.

Categories/Labels

Hotfix Any change that is a fix to an issue that blocks developers from doing their daily work, for example CI breakage, Test breakage, Minor documentation fixes that impact the user experience.

Such fixes can be merged at any time after they have passed CI checks. Depending on the fix, severity, and availability of someone to review them (other than the author) they can be merged with justification without review by one of the project owners.

Trivial Trivial changes are those that appear obvious enough and do not require maintainer or code-owner involvement. Such changes should not change the logic or the design of a subsystem or component. For example a trivial change can be:

- Documentation changes
- Configuration changes
- Minor Build System tweaks
- Minor optimization to code logic without changing the logic

- Test changes and fixes
- Sample modifications to support additional configuration or boards etc.

Maintainer Any changes that touch the logic or the original design of a subsystem or component will need to be reviewed by the code owner or the designated subsystem maintainer. If the code changes is initiated by a contributor or developer other than the owner the pull request needs to be assigned to the code owner who will have to drive the pull request to a mergeable state by giving feedback to the author and asking for more reviews from other developers.

Security Changes that appear to have an impact to the overall security of the system need to be reviewed by a security expert from the security working group.

TSC and Working Groups Changes that introduce new features or functionality or change the way the overall system works need to be reviewed by the TSC or the responsible Working Group. For example for *stable API changes*, the proposal needs to be presented in the API meeting so that the relevant stakeholders are made aware of the change.

A Pull-Request should have an Assignee

- An assignee to a pull request should not be the same as the author of the pull-request
- An assignee to a pull request is responsible for driving the pull request to a mergeable state
- An assignee is responsible for dismissing stale reviews and seeking reviews from additional developers and contributors
- Pull requests should not be merged without an approval by the assignee.

Pull Request should not be merged by author without review

All pull requests need to be reviewed and should not be merged by the author without a review. The following exceptions apply:

- Hot fixes: Fixing CI issues, reverts, and system breakage
- Release related changes: Changing version file, applying tags and release related activities without any code changes.

Developers and contributors should always seek review, however there are cases when reviewers are not available and there is a need to get a code change into the tree as soon as possible.

Reviewers shall not 'Request Changes' without comments or justification

Any change requests (-1) on a pull request have to be justified. A reviewer should avoid blocking a pull-request with no justification. If a reviewer feels that a change should not be merged without their review, then: Request change of the category: for example:

- Trivial -> Maintainer
- Assign Pull Request to yourself, this will mean that a pull request should not be merged without your approval.

Pull Requests should have at least 2 approvals before they are merged

A pull-request shall be merged only with two positive reviews (approval). Beside the person merging the pull-request (merging != approval), two additional approvals are required to be able to merge a pull request. The person merging the request can merge without approving or approve and merge to get to the 2 approvals required.

Reviewers should keep track of pull requests they have provided feedback to

If a reviewer has requested changes in a pull request, he or she should monitor the state of the pull request and/or respond to mention requests to see if his feedback has been addressed. Failing to do so, negative reviews shall be dismissed by the assignee or an owner of the repository. Reviews will be dismissed following the criteria below:

- The feedback or concerns were visibly addressed by the author
- The reviewer did not revisit the pull request after 2 week and multiple pings by the author
- The review is unrelated to the code change or asking for unjustified structural changes such as:
  - Split the PR
  - Split the commits
  - Can you fix this unrelated code that happens to appear in the diff
  - Can you fix unrelated issues
  - Etc.

Closing Stale Issues and Pull Requests

- The Pull requests and issues sections on Github are NOT discussion forums. They are items that we need to execute and drive to closure. Use the mailing lists for discussions.
- In case of both issues and pull-requests the original poster needs to respond to questions and provide clarifications regarding the issue or the change. After one week without a response to a request, a second attempt to elicit a response from the contributor will be made. After one more week without a response the item may be closed (draft and DNM tagged pull requests are excluded).

4.6.2 Continuous Integration

All changes submitted to GitHub are subject to tests that are run on emulated platforms and architectures to identify breakage and regressions that can be immediately identified. Testing using Twister additionally performs build tests of all boards and platforms. Documentation changes are also verified through review and build testing to verify doc generation will be successful.

Any failures found during the CI test run will result in a negative review assigned automatically by the CI system. Developers are expected to fix issues and rework their patches and submit again.

The CI infrastructure currently runs the following tests:

- Run “checkpatch” for code style issues (can vote -1 on errors; see note)
- Gitlint: Git commit style based on project requirements
- License Check: Check for conflicting licenses
- Run “twister” script
  - Run kernel tests in QEMU (can vote -1 on errors)
  - Build various samples for different boards (can vote -1 on errors)

- Verify documentation builds correctly.

Note: "checkpatch" is a Perl script that uses regular expressions to extract information that requires a C language parser to process accurately. As such it sometimes issues false positives. Known cases include constructs like:

```
static uint8_t __aligned(PAGE_SIZE) page_pool[PAGE_SIZE * POOL_PAGES];
IOPCTL_Type *base = config->base;
```

Both lines produce a diagnostic regarding spaces around the `*` operator: the first is misidentifed as a pointer type declaration that would be correct as `PAGE_SIZE *POOL_PAGES` while the second is misidentified as a multiplication expression that would be correct as `IOPCTL_Type * base`.

Maintainers can override the -1 in cases where the CI infrastructure gets the wrong answer.

4.6.3 Labeling issues and pull requests in GitHub

The project uses GitHub issues and pull requests (PRs) to track and manage daily and long-term work and contributions to the Zephyr project. We use GitHub **labels** to classify and organize these issues and PRs by area, type, priority, and more, making it easier to find and report on relevant items.

All GitHub issues or pull requests must be appropriately labeled. Issues and PRs often have multiple labels assigned, to help classify them in the different available categories. When reviewing a PR, if it has missing or incorrect labels, maintainers shall fix it.

This saves us all time when searching, reduces the chances of the PR or issue being forgotten, speeds up reviewing, avoids duplicate issue reports, etc.

These are the labels we currently have, grouped by type:

Area

Labels	`Area:*`
Applicable to	PRs and issues
Description	Indicates subsystems (e.g., Kernel, I2C, Memory Management), project functions (e.g., Debugging, Documentation, Process), or other categories (e.g., Coding Style, MISRA-C) affected by the bug or pull request.

An area maintainer should be able to filter by an area label and find all issues and PRs which relate to that area.

Platform

Labels	`Platform:*`
Applicable to	PRs and issues
Description	An issue or PR which affects only a particular platform

To be discussed in a meeting

Labels	`dev-review`, `TSC`
Applicable to	PRs and issues
Description	The issue is to be discussed in the following dev-review/TSC meeting if time permits

Stable API changes

Labels	`Stable API Change`
Applicable to	PRs and issues
Description	The issue or PR describes a change to a stable API. See additional information in *Introducing incompatible changes*

Minimum PR review time

Labels	`Hot Fix`, `Trivial`, `Maintainer`, `Security Review`, `TSC`
Applicable to	PRs only
Description	Depending on the PR complexity, an indication of how long a merge should be held to ensure proper review. See *review process*

Issue priority labels

Labels	`priority:{high|medium|low}`
Applicable to	Issues only
Description	To classify the impact and importance of a bug or *feature*

Note: Issue priorities are generally set or changed during the bug-triage or TSC meetings.

Miscellaneous labels

For both PRs and issues

`Bug`	The issue is a bug, or the PR is fixing a bug
`Coverity`	A Coverity detected issue or its fix
`Waiting for response`	The Zephyr developers are waiting for the submitter to respond to a question, or address an issue.
`Blocked`	Blocked by another PR or issue
`In progress`	For PRs: is work in progress and should not be merged yet. For issues: Is being worked on
`RFC`	The author would like input from the community. For a PR it should be considered a draft
`LTS`	Long term release branch related
`EXT`	Related to an external component (in `ext/`)

PR only labels

`DNM`	This PR should not be merged (Do Not Merge). For work in progress, GitHub "draft" PRs are preferred
`Stale PR`	PR which seems abandoned, and requires attention by the author
`Needs review`	The PR needs attention from the maintainers
`Backport`	The PR is a backport or should be backported
`Licensing`	The PR has licensing issues which require a licensing expert to review it

Issue only labels

`Regression`	Something, which was working, but does not anymore (bug subtype)
`Question`	This issue is a question to the Zephyr developers
`Enhancement`	Changes/Updates/Additions to existing *features*
`Feature request`	A request for a new *feature*
`Feature`	A *planned feature* with a milestone
`Duplicate`	This issue is a duplicate of another issue (please specify)
`Good first issue`	Good for a first time contributor to take
`Release Notes`	Issues that need to be mentioned in release notes as known issues with additional information

Any issue must be classified and labeled as either `Bug`, `Question`, `Enhancement`, `Feature`, or `Feature Request`. More information on how feature requests are handled and become features can be found in *Feature Tracking*.

4.7 Bug Reporting

To maintain traceability and relation between proposals, changes, features, and issues, it is recommended to cross-reference source code commits with the relevant GitHub issues and vice versa. Any changes that originate from a tracked feature or issue should contain a reference to the feature by mentioning the corresponding issue or pull-request identifiers.

At any time it should be possible to establish the origin of a change and the reason behind it by following the references in the code.

4.7.1 Reporting a regression issue

It could happen that the issue being reported is identified as a regression, as the use case is known to be working on earlier commit or release. In this case, providing directly the guilty commit when submitting the bug gains a lot of time in the eventual bug fixing.

To identify the commit causing the regression, several methods could be used, but tree bisecting method is an efficient one that doesn't require deep code expertise and can be used by every one.

For this, git bisect is the recommended tool.

Recommendations on the process:

- Run `west update` on each bisection step.
- Once the bisection is over and a culprit identifed, verify manually the result.

4.8 Communication and Collaboration

The Zephyr project mailing lists are used as the primary communication tool by project members, contributors, and the community. The mailing list is open for topics related to the project and should be used for collaboration among team members working on the same feature or subsystem or for discussing project direction and daily development of the code base. In general, bug reports and issues should be entered and tracked in the bug tracking system (GitHub Issues) and not broadcasted to the mailing list, the same applies to code reviews. Code should be submitted to GitHub using the appropriate tools.

4.9 Code Documentation

4.9.1 API Documentation

Well documented APIs enhance the experience for developers and are an essential requirement for defining an API's success. Doxygen is a general purpose documentation tool that the zephyr project uses for documenting APIs. It generates either an on-line documentation browser (in HTML) and/or provides input for other tools that is used to generate a reference manual from documented source files. In particular, doxygen's XML output is used as an input when producing the Zephyr project's online documentation.

4.9.2 Reference to Requirements

APIs for the most part document the implementation of requirements or advertised features and can be traced back to features. We use the API documentation as the main interface to trace implementation back to documented features. This is done using custom _doxygen_ tags that reference requirements maintained somewhere else in a requirement catalogue.

4.9.3 Test Documentation

To help understand what each test does and which functionality it tests we also document all test code using the same tools and in the same context and generate documentation for all unit and integration tests maintained in the same environment. Tests are documented using references to the APIs or functionality they validate by creating a link back to the APIs and by adding a reference to the original requirements.

4.9.4 Documentation Guidelines

Test Code

The Zephyr project uses several test methodologies, the most common being the *Ztest framework*. Test documentation should only be done on the entry test functions (usually prefixed with test_) and those that are called directly by the Ztest framework. Those tests are going to appear in test reports and using their name and identifier is the best way to identify them and trace back to them from requirements.

Test documentation should not interfere with the actual API documentation and needs to follow a new structure to avoid confusion. Using a consistent naming scheme and following a well-defined structure we will be able to group this documentation in its own module and identify it uniquely when parsing test data for traceability reports. Here are a few guidelines to be followed:

- All test code documentation should be grouped under the `all_tests` doxygen group
- All test documentation should be under doxygen groups that are prefixed with tests_

The custom doxygen `@verify` directive signifies that a test verifies a requirement:

```
/**
* @brief Tests for the Semaphore kernel object
* @defgroup kernel_semaphore_tests Semaphore
* @ingroup all_tests
* @{
*/

...
/**
* @brief A brief description of the tests
* Some details about the test
* more details
*
* @verify{@req{1111}}
*/
void test_sema_thread2thread(void)
{
...
}
...

/**
* @}
*/
```

To get coverage of how an implementation or a piece of code satisfies a requirements, we use the `satisfy` alias in doxygen:

```
/**
* @brief Give a semaphore.
*
* This routine gives @a sem, unless the semaphore is already at its maximum
* permitted count.
*
* @note Can be called by ISRs.
*
* @param sem Address of the semaphore.
*
* @return N/A
* @satisfy{@req{015}}
*/
__syscall void k_sem_give(struct k_sem *sem);
```

To generate the matrix, you will first need to build the documentation, specifically you will need to build the doxygen XML output:

```
$ make doxygen
```

Parse the generated XML data from doxygen to generate the traceability matrix.

The Zephyr project defines a development process workflow using GitHub **Issues** to track feature, enhancement, and bug reports together with GitHub **Pull Requests** (PRs) for submitting and reviewing changes. Zephyr community members work together to review these Issues and PRs, managing feature enhancements and quality improvements of Zephyr through its regular releases, as outlined in the program management overview.

We can only manage the volume of Issues and PRs, by requiring timely reviews, feedback, and responses from the community and contributors, both for initial submissions and for followup questions and clarifications. Read about the project's *development processes and tools* and specifics about *review timelines* to

learn about the project's goals and guidelines for our active developer community.

TSC Project Roles describes in detail the Zephyr project roles and associated permissions with respect to the development process workflow.

4.10 Terminology

- mainline: The main tree where the core functionality and core features are being developed.
- subsystem/feature branch: is a branch within the same repository. In our case, we will use the term branch also when referencing branches not in the same repository, which are a copy of a repository sharing the same history.
- upstream: A parent branch the source code is based on. This is the branch you pull from and push to, basically your upstream.
- LTS: Long Term Support

Chapter 5

Build and Configuration Systems

5.1 Build System (CMake)

CMake is used to build your application together with the Zephyr kernel. A CMake build is done in two stages. The first stage is called **configuration**. During configuration, the CMakeLists.txt build scripts are executed. After configuration is finished, CMake has an internal model of the Zephyr build, and can generate build scripts that are native to the host platform.

CMake supports generating scripts for several build systems, but only Ninja and Make are tested and supported by Zephyr. After configuration, you begin the **build** stage by executing the generated build scripts. These build scripts can recompile the application without involving CMake following most code changes. However, after certain changes, the configuration step must be executed again before building. The build scripts can detect some of these situations and reconfigure automatically, but there are cases when this must be done manually.

Zephyr uses CMake's concept of a 'target' to organize the build. A target can be an executable, a library, or a generated file. For application developers, the library target is the most important to understand. All source code that goes into a Zephyr build does so by being included in a library target, even application code.

Library targets have source code, that is added through CMakeLists.txt build scripts like this:

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

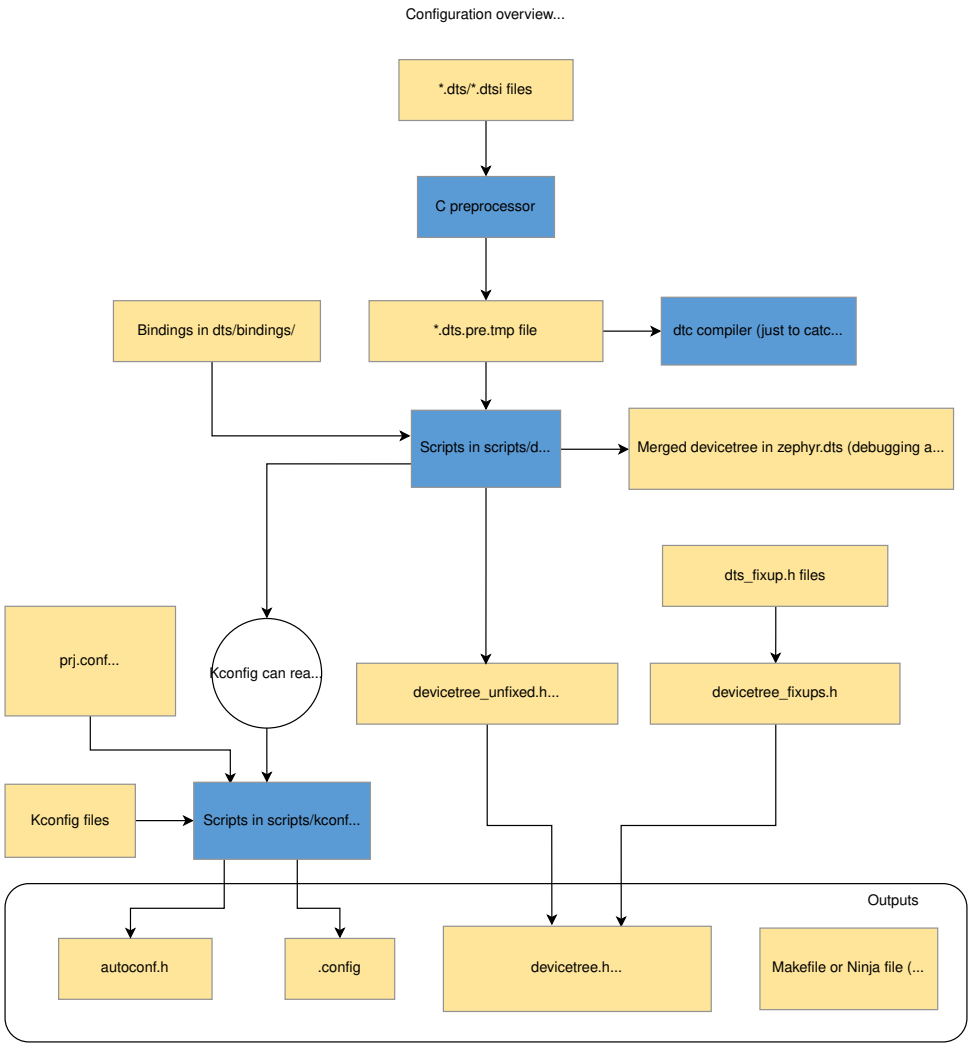
In the above `CMakeLists.txt`, an existing library target named `app` is configured to include the source file `src/main.c`. The `PRIVATE` keyword indicates that we are modifying the internals of how the library is being built. Using the keyword `PUBLIC` would modify how other libraries that link with app are built. In this case, using `PUBLIC` would cause libraries that link with `app` to also include the source file `src/main.c`, behavior that we surely do not want. The `PUBLIC` keyword could however be useful when modifying the include paths of a target library.

5.1.1 Build and Configuration Phases

The Zephyr build process can be divided into two main phases: a configuration phase (driven by CMake) and a build phase (driven by Make or Ninja).

Configuration Phase

The configuration phase begins when the user invokes *CMake*, specifying a source application directory and a board target.


Configuration overview...
.dts/.dtsi files
C preprocessor
Bindings in dts/bindings/
*.dts.pre.tmp file
dtc compiler (just to catc...
Scripts in scripts/d...
Merged devicetree in zephyr.dts (debugging a...
dts_fixup.h files
prj.conf...
Kconfig can rea...
devicetree_unfixed.h...
devicetree_fixups.h
Kconfig files
Scripts in scripts/kconf...
Outputs
autoconf.h
.config
devicetree.h...
Makefile or Ninja file (...

CMake begins by processing the `CMakeLists.txt` file in the application directory, which refers to the `CMakeLists.txt` file in the Zephyr top-level directory, which in turn refers to `CMakeLists.txt` files throughout the build tree (directly and indirectly). Its primary output is a set of Makefiles or Ninja files to drive the build process, but the CMake scripts also do some processing of their own:

Devicetree `*.dts` (*devicetree source*) and `*.dtsi` (*devicetree source include*) files are collected from the target's architecture, SoC, board, and application directories.

`*.dtsi` files are included by `*.dts` files via the C preprocessor (often abbreviated *cpp*, which should not be confused with C++). The C preprocessor is also used to merge in any devicetree `*.overlay` files, and to expand macros in `*.dts`, `*.dtsi`, and `*.overlay` files.

The preprocessed devicetree sources (stored in `*.dts.pre.tmp`) are parsed by gen_defines.py to generate a `devicetree_unfixed.h` header with preprocessor macros.

As a debugging aid, `gen_defines.py` writes the final devicetree to `zephyr.dts`. This file is just for reference. It is not used anywhere.

The `dtc` devicetree compiler also gets run on the preprocessed devicetree sources to catch any extra warnings and errors generated by it. The output from `dtc` is unused otherwise.

The above is just a brief overview. For more information on devicetree, see *Devicetree Guide*.

Devicetree fixups Files named `dts_fixup.h` from the target's architecture, SoC, board, and application directories are concatenated into a single `devicetree_fixups.h` file. `dts_fixup.h` files are used to rename generated macros to names expected by the source code.

Source code accesses preprocessor macros generated from devicetree by including the devicetree.h header, which includes `devicetree_unfixed.h` and `devicetree_fixups.h`.

Kconfig `Kconfig` files define available configuration options for for the target architecture, SoC, board, and application, as well as dependencies between options.

Kconfig configurations are stored in *configuration files*. The initial configuration is generated by merging configuration fragments from the board and application (e.g. `prj.conf`).

The output from Kconfig is an `autoconf.h` header with preprocessor assignments, and a `.config` file that acts both as a saved configuration and as configuration output (used by CMake).

Information from devicetree is available to Kconfig, through the functions defined in kconfigfunctions.py.

See *the Kconfig section of the manual* for more information.

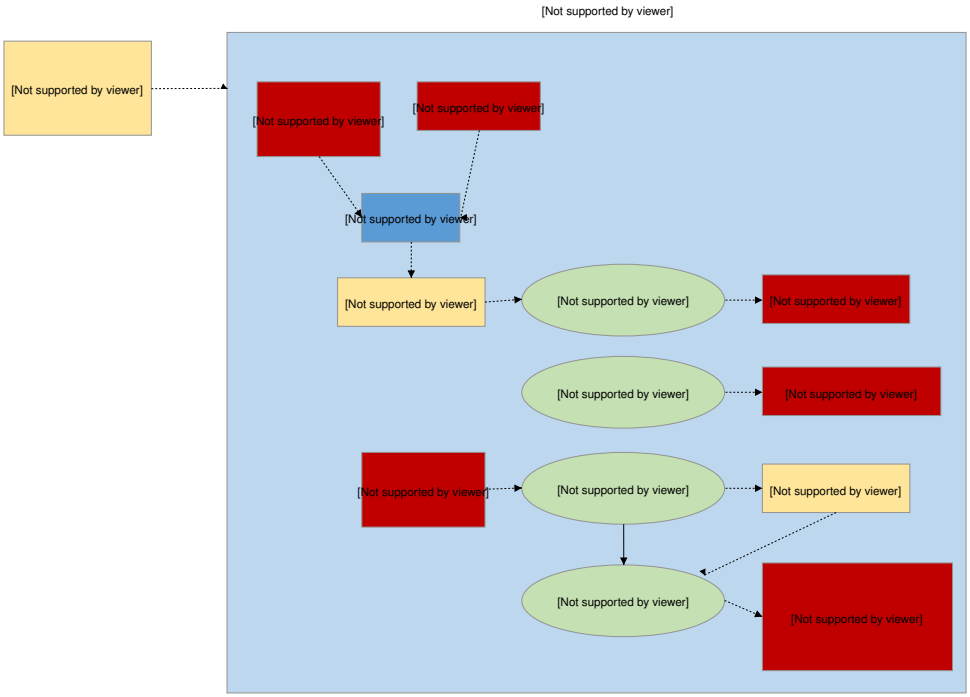
Build Phase

The build phase begins when the user invokes `make` or `ninja`. Its ultimate output is a complete Zephyr application in a format suitable for loading/flashing on the desired target board (`zephyr.elf`, `zephyr.hex`, etc.) The build phase can be broken down, conceptually, into four stages: the pre-build, first-pass binary, final binary, and post-processing.

Pre-build Pre-build occurs before any source files are compiled, because during this phase header files used by the source files are generated.

Offset generation Access to high-level data structures and members is sometimes required when the definitions of those structures is not immediately accessible (e.g., assembly language). The generation of *offsets.h* (by *gen_offset_header.py*) facilitates this.

System call boilerplate The *gen_syscall.py* and *parse_syscalls.py* scripts work together to bind potential system call functions with their implementations.



First-pass binary Compilation proper begins with the first-pass binary. Source files (C and assembly) are collected from various subsystems (which ones is decided during the configuration phase), and compiled into archives (with reference to header files in the tree, as well as those generated during the configuration phase and the pre-build stage).

If memory protection is enabled, then:

Partition grouping The *gen_app_partitions.py* script scans all the generated archives and outputs linker scripts to ensure that application partitions are properly grouped and aligned for the target's memory protection hardware.

Then *cpp* is used to combine linker script fragments from the target's architecture/SoC, the kernel tree, optionally the partition output if memory protection is enabled, and any other fragments selected during the configuration process, into a *linker.cmd* file. The compiled archives are then linked with *ld* as specified in the *linker.cmd*.

In some configurations, this is the final binary, and the next stage is skipped.

Final binary The binary from the previous stage is incomplete, with empty and/or placeholder sections that must be filled in by, essentially, reflection.

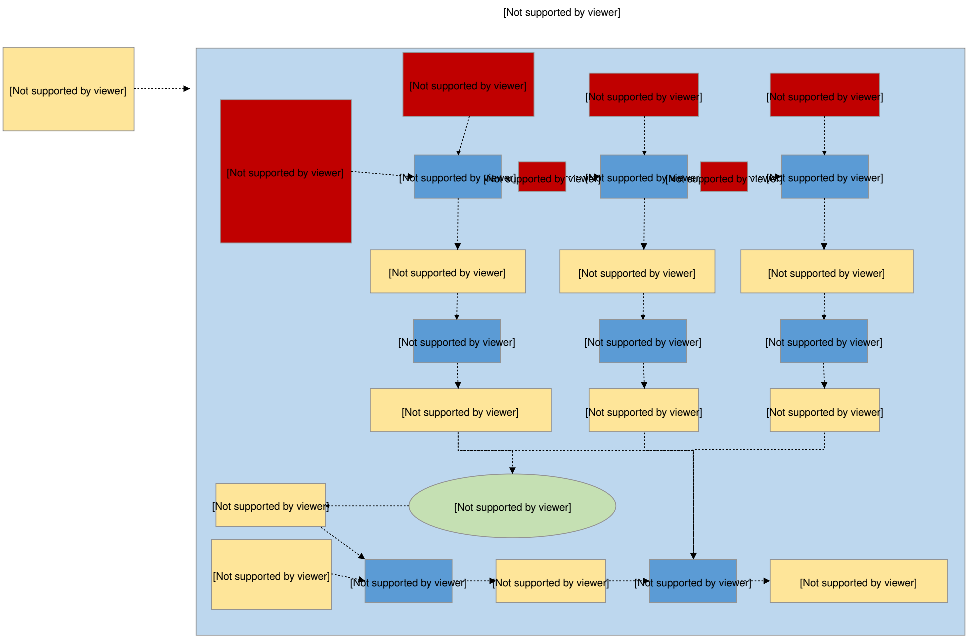
Device dependencies The *gen_handles.py* script scans the first-pass binary to determine relationships between devices that were recorded from devicetree data, and replaces the encoded relationships with values that are optimized to locate the devices actually present in the application.

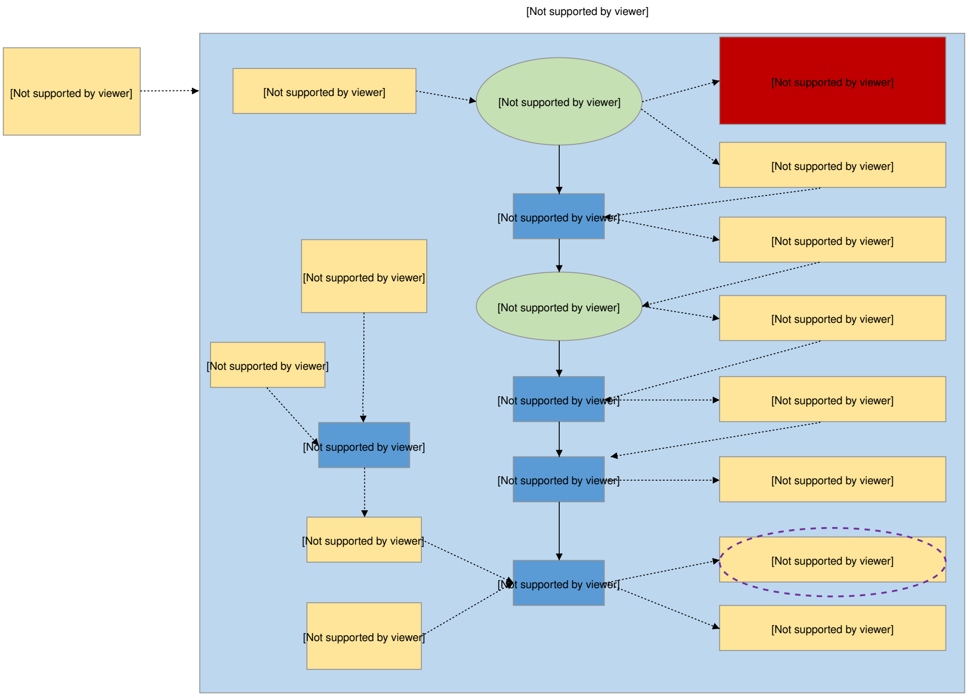
When *User Mode* is enabled:

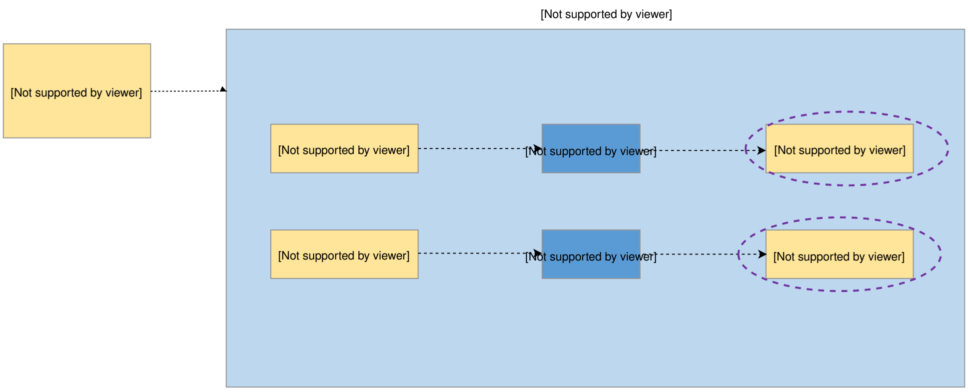
Kernel object hashing The *gen_kobject_list.py* scans the *ELF DWARF* debug data to find the address of the all kernel objects. This list is passed to *gperf*, which generates a perfect hash function and table of those addresses, then that output is optimized by *process_gperf.py*, using known properties of our special case.

Then, the link from the previous stage is repeated, this time with the missing pieces populated.

Post processing Finally, if necessary, the completed kernel is converted from *ELF* to the format expected by the loader and/or flash tool required by the target. This is accomplished in a straightforward manner with *objdump*.


[Not supported by viewer]


[Not supported by viewer]


[Not supported by viewer]

5.1.2 Supporting Scripts and Tools

The following is a detailed description of the scripts used during the build process.

scripts/gen_syscalls.py

Script to generate system call invocation macros

This script parses the system call metadata JSON file emitted by parse_syscalls.py to create several files:

- A file containing weak aliases of any potentially unimplemented system calls, as well as the system call dispatch table, which maps system call type IDs to their handler functions.
- A header file defining the system call type IDs, as well as function prototypes for all system call handler functions.
- A directory containing header files. Each header corresponds to a header that was identified as containing system call declarations. These generated headers contain the inline invocation functions for each system call in that header.

scripts/gen_handles.py

Translate generic handles into ones optimized for the application.

Immutable device data includes information about dependencies, e.g. that a particular sensor is controlled through a specific I2C bus and that it signals event on a pin on a specific GPIO controller. This information is encoded in the first-pass binary using identifiers derived from the devicetree. This script extracts those identifiers and replaces them with ones optimized for use with the devices actually present.

For example the sensor might have a first-pass handle defined by its devicetree ordinal 52, with the I2C driver having ordinal 24 and the GPIO controller ordinal 14. The runtime ordinal is the index of the corresponding device in the static devicetree array, which might be 6, 5, and 3, respectively.

The output is a C source file that provides alternative definitions for the array contents referenced from the immutable device objects. In the final link these definitions supersede the ones in the driver-specific object file.

scripts/gen_kobject_list.py

Script to generate gperf tables of kernel object metadata

User mode threads making system calls reference kernel objects by memory address, as the kernel/driver APIs in Zephyr are the same for both user and supervisor contexts. It is necessary for the kernel to be able to validate accesses to kernel objects to make the following assertions:

- That the memory address points to a kernel object
- The kernel object is of the expected type for the API being invoked
- The kernel object is of the expected initialization state
- The calling thread has sufficient permissions on the object

For more details see the *Kernel Objects* section in the documentation.

The zephyr build generates an intermediate ELF binary, zephyr_prebuilt.elf, which this script scans looking for kernel objects by examining the DWARF debug information to look for instances of data structures that are considered kernel objects. For device drivers, the API struct pointer populated at build time is also examined to disambiguate between various device driver instances since they are all 'struct device'.

This script can generate five different output files:

- A gperf script to generate the hash table mapping kernel object memory addresses to kernel object metadata, used to track permissions, object type, initialization state, and any object-specific data.
- A header file containing generated macros for validating driver instances inside the system call handlers for the driver subsystem APIs.
- A code fragment included by kernel.h with one enum constant for each kernel object type and each driver instance.
- The inner cases of a switch/case C statement, included by kernel/userspace.c, mapping the kernel object types and driver instances to their human-readable representation in the otype_to_str() function.
- The inner cases of a switch/case C statement, included by kernel/userspace.c, mapping kernel object types to their sizes. This is used for allocating instances of them at runtime (CONFIG_DYNAMIC_OBJECTS) in the obj_size_get() function.

scripts/gen_offset_header.py

This script scans a specified object file and generates a header file that defined macros for the offsets of various found structure members (particularly symbols ending with `_OFFSET` or `_SIZEOF`), primarily intended for use in assembly code.

scripts/parse_syscalls.py

Script to scan Zephyr include directories and emit system call and subsystem metadata

System calls require a great deal of boilerplate code in order to implement completely. This script is the first step in the build system's process of auto-generating this code by doing a text scan of directories containing C or header files, and building up a database of system calls and their function call prototypes. This information is emitted to a generated JSON file for further processing.

This script also scans for struct definitions such as __subsystem and __net_socket, emitting a JSON dictionary mapping tags to all the struct declarations found that were tagged with them.

If the output JSON file already exists, its contents are checked against what information this script would have outputted; if the result is that the file would be unchanged, it is not modified to prevent unnecessary incremental builds.

arch/x86/gen_idt.py

Generate Interrupt Descriptor Table for x86 CPUs.

This script generates the interrupt descriptor table (IDT) for x86. Please consult the IA Architecture SW Developer Manual, volume 3, for more details on this data structure.

This script accepts as input the zephyr_prebuilt.elf binary, which is a link of the Zephyr kernel without various build-time generated data structures (such as the IDT) inserted into it. This kernel image has been properly padded such that inserting these data structures will not disturb the memory addresses of other symbols. From the kernel binary we read a special section "intList" which contains the desired interrupt routing configuration for the kernel, populated by instances of the IRQ_CONNECT() macro.

This script outputs three binary tables:

1. The interrupt descriptor table itself.
2. A bitfield indicating which vectors in the IDT are free for installation of dynamic interrupts at runtime.
3. An array which maps configured IRQ lines to their associated vector entries in the IDT, used to program the APIC at runtime.

arch/x86/gen_gdt.py

Generate a Global Descriptor Table (GDT) for x86 CPUs.

For additional detail on GDT and x86 memory management, please consult the IA Architecture SW Developer Manual, vol. 3.

This script accepts as input the zephyr_prebuilt.elf binary, which is a link of the Zephyr kernel without various build-time generated data structures (such as the GDT) inserted into it. This kernel image has been properly padded such that inserting these data structures will not disturb the memory addresses of other symbols.

The input kernel ELF binary is used to obtain the following information:

- Memory addresses of the Main and Double Fault TSS structures so GDT descriptors can be created for them
- Memory addresses of where the GDT lives in memory, so that this address can be populated in the GDT pseudo descriptor
- whether userspace or HW stack protection are enabled in Kconfig

The output is a GDT whose contents depend on the kernel configuration. With no memory protection features enabled, we generate flat 32-bit code and data segments. If hardware- based stack overflow protection or userspace is enabled, we additionally create descriptors for the main and double- fault IA tasks, needed for userspace privilege elevation and double-fault handling. If userspace is enabled, we also create flat code/data segments for ring 3 execution.

scripts/gen_relocate_app.py

This script will relocate .text, .rodata, .data and .bss sections from required files and places it in the required memory region. This memory region and file are given to this python script in the form of a string.

Example of such a string would be:

```
SRAM2:/home/xyz/zephyr/samples/hello_world/src/main.c,\
SRAM1:/home/xyz/zephyr/samples/hello_world/src/main2.c
```

To invoke this script:

```
python3 gen_relocate_app.py -i input_string -o generated_linker -c generated_code
```

Configuration that needs to be sent to the python script.

- If the memory is like SRAM1/SRAM2/CCD/AON then place full object in the sections
- If the memory type is appended with _DATA / _TEXT/ _RODATA/ _BSS only the selected memory is placed in the required memory region. Others are ignored.

Multiple regions can be appended together like SRAM2_DATA_BSS this will place data and bss inside SRAM2.

scripts/process_gperf.py

gperf C file post-processor

We use gperf to build up a perfect hashtable of pointer values. The way gperf does this is to create a table 'wordlist' indexed by a string representation of a pointer address, and then doing memcmp() on a string passed in for comparison

We are exclusively working with 4-byte pointer values. This script adjusts the generated code so that we work with pointers directly and not strings. This saves a considerable amount of space.

scripts/gen_app_partitions.py

Script to generate a linker script organizing application memory partitions

Applications may declare build-time memory domain partitions with K_APPMEM_PARTITION_DEFINE, and assign globals to them using K_APP_DMEM or K_APP_BMEM macros. For each of these partitions, we need to route all their data into appropriately-sized memory areas which meet the size/alignment constraints of the memory protection hardware.

This linker script is created very early in the build process, before the build attempts to link the kernel binary, as the linker script this tool generates is a necessary pre-condition for kernel linking. We extract the set of memory partitions to generate by looking for variables which have been assigned to input sections that follow a defined naming convention. We also allow entire libraries to be pulled in to assign their globals to a particular memory partition via command line directives.

This script takes as inputs:

- The base directory to look for compiled objects
- key/value pairs mapping static library files to what partitions their globals should end up in.

The output is a linker script fragment containing the definition of the app shared memory section, which is further divided, for each partition found, into data and BSS for each partition.

5.2 Configuration System (Kconfig)

The Zephyr kernel and subsystems can be configured at build time to adapt them for specific application and platform needs. Configuration is handled through Kconfig, which is the same configuration system used by the Linux kernel. The goal is to support configuration without having to change any source code.

Configuration options (often called *symbols*) are defined in `Kconfig` files, which also specify dependencies between symbols that determine what configurations are valid. Symbols can be grouped into menus and sub-menus to keep the interactive configuration interfaces organized.

The output from Kconfig is a header file `autoconf.h` with macros that can be tested at build time. Code for unused features can be compiled out to save space.

The following sections explain how to set Kconfig configuration options, go into detail on how Kconfig is used within the Zephyr project, and have some tips and best practices for writing `Kconfig` files.

5.2.1 Interactive Kconfig interfaces

There are two interactive configuration interfaces available for exploring the available Kconfig options and making temporary changes: `menuconfig` and `guiconfig`. `menuconfig` is a curses-based interface that runs in the terminal, while `guiconfig` is a graphical configuration interface.

Note: The configuration can also be changed by editing `zephyr/.config` in the application build directory by hand. Using one of the configuration interfaces is often handier, as they correctly handle dependencies between configuration symbols.

If you try to enable a symbol with unsatisfied dependencies in `zephyr/.config`, the assignment will be ignored and overwritten when re-configuring.

To make a setting permanent, you should set it in a `*.conf` file, as described in *Setting Kconfig configuration values*.

Tip: Saving a minimal configuration file (with e.g. `D` in menuconfig) and inspecting it can be handy when making settings permanent. The minimal configuration file only lists symbols that differ from their

default value.

To run one of the configuration interfaces, do this:

1. Build your application as usual using either `west` or `cmake`:

   Using west:

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

   Using CMake and ninja:

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

2. To run the terminal-based `menuconfig` interface, use either of these commands:

   ```
   west build -t menuconfig
   ```

   ```
   ninja menuconfig
   ```

   To run the graphical `guiconfig`, use either of these commands:

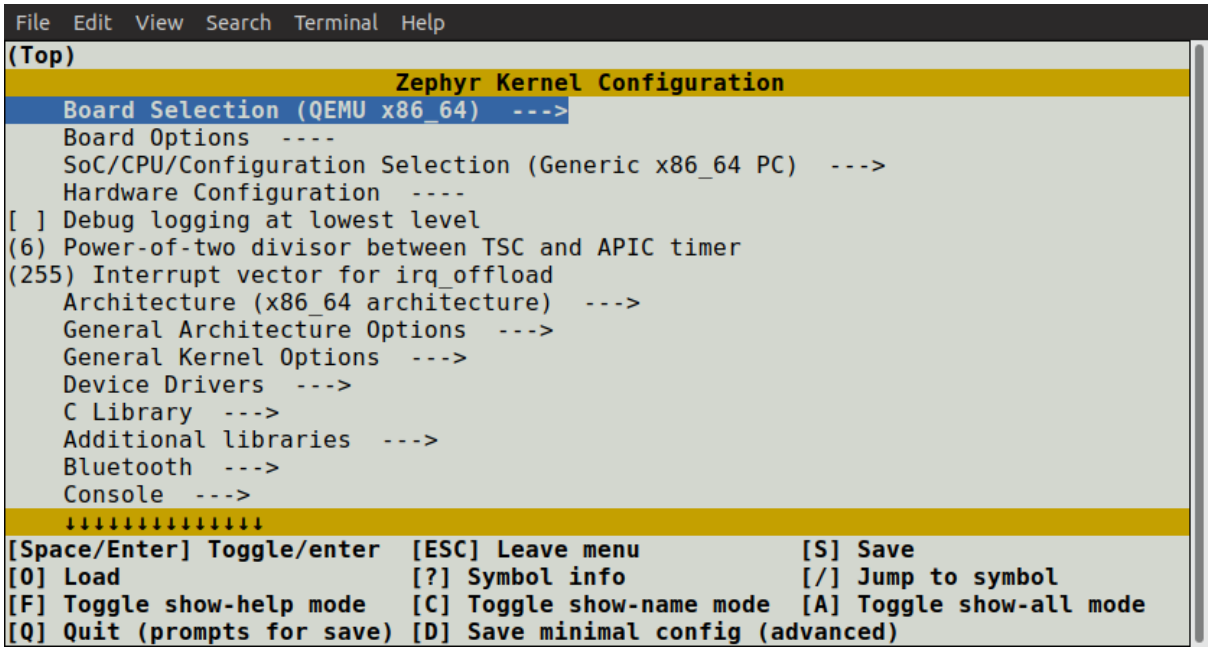
   ```
   west build -t guiconfig
   ```

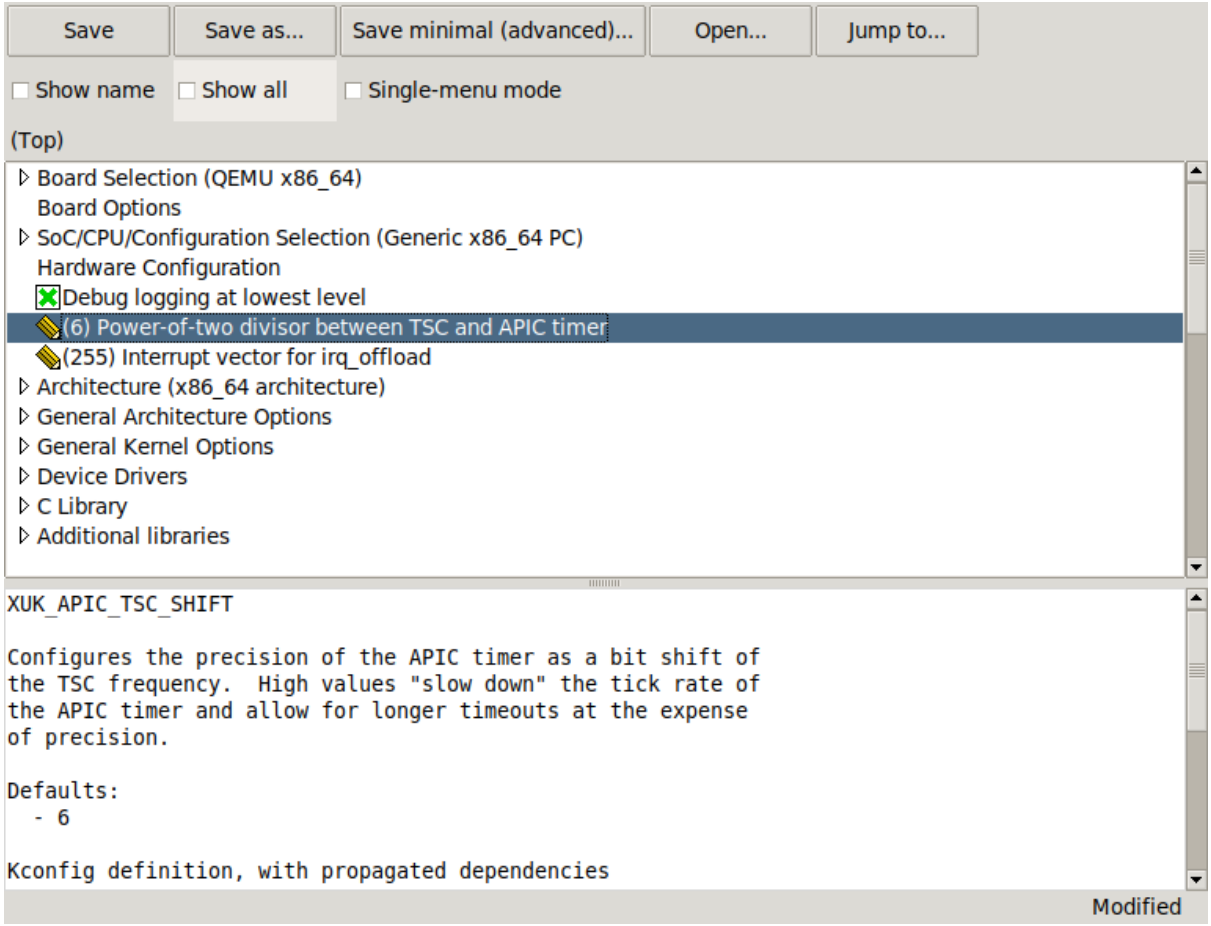
   ```
   ninja guiconfig
   ```

   Note: If you get an import error for `tkinter` when trying to run `guiconfig`, you are missing required packages. See *Install Linux Host Dependencies*. The package you need is usually called something like `python3-tk/python3-tkinter`.

   `tkinter` is not included by default in many Python installations, despite being part of the standard library.

   The two interfaces are shown below:





`guiconfig` always shows the help text and other information related to the currently selected item in the bottom window pane. In the terminal interface, press `?` to view the same information.

Note: If you prefer to work in the `guiconfig` interface, then it's a good idea to check any changes to Kconfig files you make in *single-menu mode*, which is toggled via a checkbox at the top. Unlike full-tree mode, single-menu mode will distinguish between symbols defined with `config` and symbols defined with `menuconfig`, showing you what things would look like in the `menuconfig` interface.

---

3. Change configuration values in the `menuconfig` interface as follows:

   - Navigate the menu with the arrow keys. Common Vim key bindings are supported as well.
   - Use `Space` and `Enter` to enter menus and toggle values. Menus appear with `--->` next to them. Press `ESC` to return to the parent menu.

     Boolean configuration options are shown with *[ ]* brackets, while numeric and string-valued configuration symbols are shown with *( )* brackets. Symbol values that can't be changed are shown as `- -` or `-*-`.

     Note: You can also press `Y` or `N` to set a boolean configuration symbol to the corresponding value.

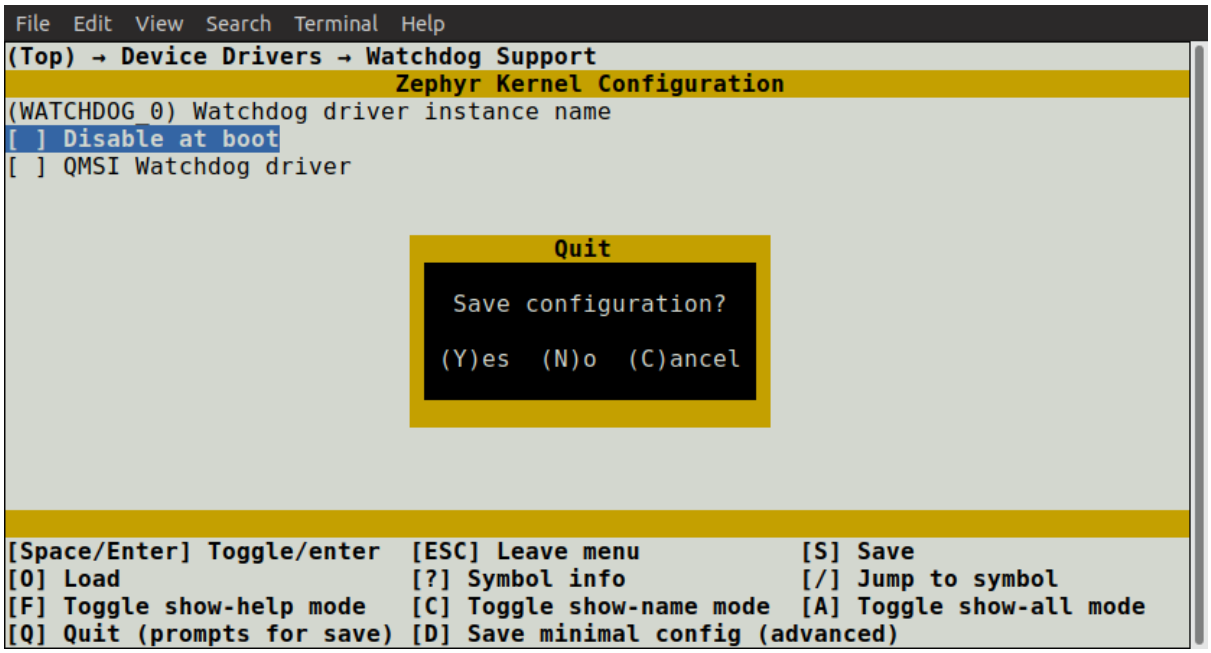
   - Press `?` to display information about the currently selected symbol, including its help text. Press `ESC` or `Q` to return from the information display to the menu.

   In the `guiconfig` interface, either click on the image next to the symbol to change its value, or

double-click on the row with the symbol (this only works if the symbol has no children, as double-clicking a symbol with children open/closes its menu instead).

`guiconfig` also supports keyboard controls, which are similar to `menuconfig`.

4. Pressing `Q` in the `menuconfig` interface will bring up the save-and-quit dialog (if there are changes to save):



Press `Y` to save the kernel configuration options to the default filename (`zephyr/.config`). You will typically save to the default filename unless you are experimenting with different configurations.

The `guiconfig` interface will also prompt for saving the configuration on exit if it has been modified.

Note: The configuration file used during the build is always `zephyr/.config`. If you have another saved configuration that you want to build with, copy it to `zephyr/.config`. Make sure to back up your original configuration file.

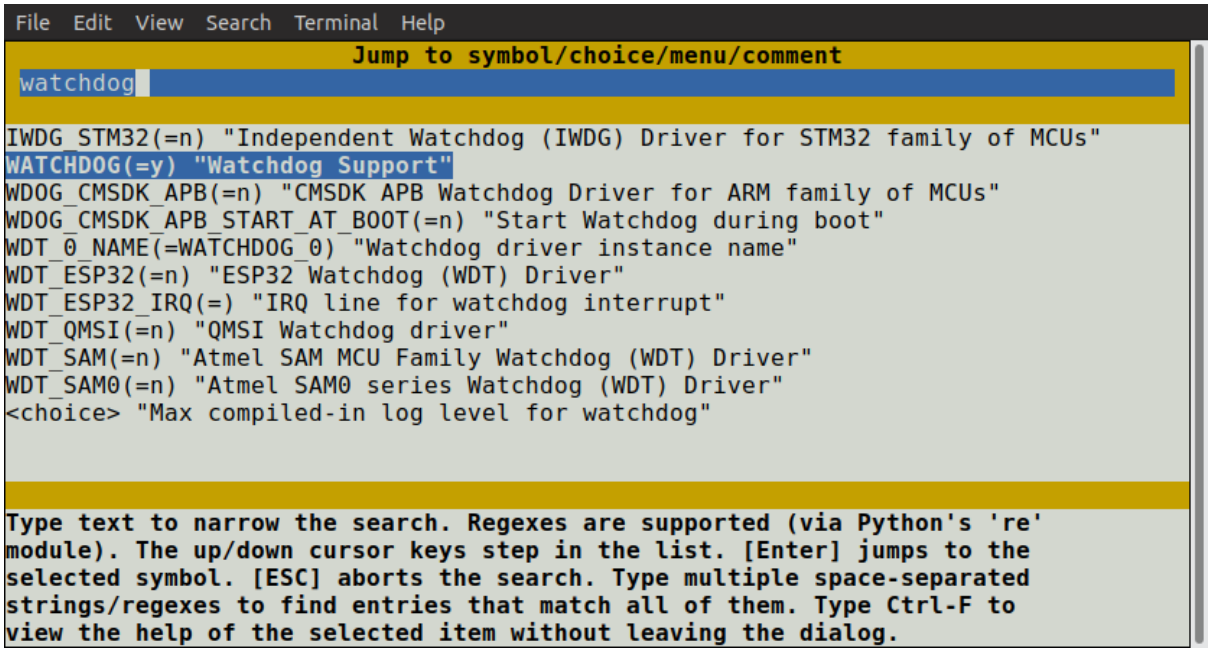
Also note that filenames starting with `.` are not listed by `ls` by default on Linux and macOS. Use the `-a` flag to see them.

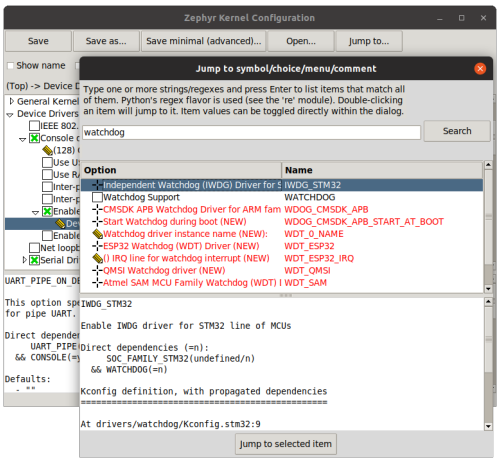
Finding a symbol in the menu tree and navigating to it can be tedious. To jump directly to a symbol, press the `/` key (this also works in `guiconfig`). This brings up the following dialog, where you can search for symbols by name and jump to them. In `guiconfig`, you can also change symbol values directly within the dialog.

If you jump to a symbol that isn't currently visible (e.g., due to having unsatisfied dependencies), then *show-all mode* will be enabled. In show-all mode, all symbols are displayed, including currently invisible symbols. To turn off show-all mode, press `A` in `menuconfig` or `Ctrl-A` in `guiconfig`.

Note: Show-all mode can't be turned off if there are no visible items in the current menu.

To figure out why a symbol you jumped to isn't visible, inspect its dependencies, either by pressing `?` in `menuconfig` or in the information pane at the bottom in `guiconfig`. If you discover that the symbol depends on another symbol that isn't enabled, you can jump to that symbol in turn to see if it can be enabled.


File Edit View Search Terminal Help
Jump to symbol/choice/menu/comment
watchdog
IWDG_STM32(=n) "Independent Watchdog (IWDG) Driver for STM32 family of MCUs"
WATCHDOG(=y) "Watchdog Support"
WDOG_CMSDK_APB(=n) "CMSDK APB Watchdog Driver for ARM family of MCUs"
WDOG_CMSDK_APB_START_AT_BOOT(=n) "Start Watchdog during boot"
WDT_0_NAME(=WATCHDOG_0) "Watchdog driver instance name"
WDT_ESP32(=n) "ESP32 Watchdog (WDT) Driver"
WDT_ESP32_IRQ(=) "IRQ line for watchdog interrupt"
WDT_QMSI(=n) "QMSI Watchdog driver"
WDT_SAM(=n) "Atmel SAM MCU Family Watchdog (WDT) Driver"
WDT_SAM0(=n) "Atmel SAM0 series Watchdog (WDT) Driver"
<choice> "Max compiled-in log level for watchdog"
Type text to narrow the search. Regexes are supported (via Python's 're' module). The up/down cursor keys step in the list. [Enter] jumps to the selected symbol. [ESC] aborts the search. Type multiple space-separated strings/regexes to find entries that match all of them. Type Ctrl-F to view the help of the selected item without leaving the dialog.


Zephyr Kernel Configuration
Save
Save as...
Save minimal (advanced)...
Open...
Jump to...
Jump to symbol/choice/menu/comment
Type one or more strings/regexes and press Enter to list items that match all of them. Python's regex flavor is used (see the 're' module). Double-clicking an item will jump to it. Item values can be toggled directly within the dialog.
watchdog
Search
Option
Name
Independent Watchdog (IWDG) Driver for S
IWDG_STM32
Watchdog Support
WATCHDOG
CMSDK APB Watchdog Driver for ARM fam
WDOG_CMSDK_APB
Start Watchdog during boot (NEW)
WDOG_CMSDK_APB_START_AT_BOOT
Watchdog driver instance name (NEW):
WDT_0_NAME
ESP32 Watchdog (WDT) Driver (NEW)
WDT_ESP32
() IRQ line for watchdog interrupt (NEW)
WDT_ESP32_IRQ
QMSI Watchdog driver (NEW)
WDT_QMSI
Atmel SAM MCU Family Watchdog (WDT) (
WDT_SAM
IWDG_STM32
Enable IWDG driver for STM32 line of MCUs
Direct dependencies (=n):
SOC_FAMILY_STM32(undefined/n)
&& WATCHDOG(=n)
Kconfig definition, with propagated dependencies
At drivers/watchdog/Kconfig.stm32:9
Jump to selected item

Note: In `menuconfig`, you can press `Ctrl-F` to view the help of the currently selected item in the jump-to dialog without leaving the dialog.

For more information on `menuconfig` and `guiconfig`, see the Python docstrings at the top of menuconfig.py and guiconfig.py.

5.2.2 Setting Kconfig configuration values

The *menuconfig and guiconfig interfaces* can be used to test out configurations during application development. This page explains how to make settings permanent.

An auto-generated list of all Kconfig options can be found in the Kconfig symbol reference.

Note: Before making changes to Kconfig files, it's a good idea to also go through the *Kconfig - Tips and Best Practices* page.

Visible and invisible Kconfig symbols

When making Kconfig changes, it's important to understand the difference between *visible* and *invisible* symbols.

- A visible symbol is a symbol defined with a prompt. Visible symbols show up in the interactive configuration interfaces (hence *visible*), and can be set in configuration files.

  Here's an example of a visible symbol:

  ```
  config FPU
     bool "Support floating point operations"
     depends on HAS_FPU
  ```

  The symbol is shown like this in `menuconfig`, where it can be toggled:

  ```
  [ ] Support floating point operations
  ```

- An *invisible* symbol is a symbol without a prompt. Invisible symbols are not shown in the interactive configuration interfaces, and users have no direct control over their value. They instead get their value from defaults or from other symbols.

  Here's an example or an invisible symbol:

  ```
  config CPU_HAS_FPU
     bool
     help
       This symbol is y if the CPU has a hardware floating point unit.
  ```

  In this case, `CPU_HAS_FPU` is enabled through other symbols having `select CPU_HAS_FPU`.

Setting symbols in configuration files

Visible symbols can be configured by setting them in configuration files. The initial configuration is produced by merging a `*_defconfig` file for the board with application settings, usually from `prj.conf`. See *The Initial Configuration* below for more details.

Assignments in configuration files use this syntax:

```
CONFIG_<symbol name>=<value>
```

There should be no spaces around the equals sign.

`bool` symbols can be enabled or disabled by setting them to `y` or `n`, respectively. The `FPU` symbol from the example above could be enabled like this:

```
CONFIG_FPU=y
```

Note: A boolean symbol can also be set to `n` with a comment formatted like this:

```
# CONFIG_SOME_OTHER_BOOL is not set
```

This is the format you will see in the merged configuration in `zephyr/.config`.

This style is accepted for historical reasons: Kconfig configuration files can be parsed as makefiles (though Zephyr doesn't use this). Having `n`-valued symbols correspond to unset variables simplifies tests in Make.

Other symbol types are assigned like this:

```
CONFIG_SOME_STRING="cool value"
CONFIG_SOME_INT=123
```

Comments use a `#`:

```
# This is a comment
```

Assignments in configuration files are only respected if the dependencies for the symbol are satisfied. A warning is printed otherwise. To figure out what the dependencies of a symbol are, use one of the *interactive configuration interfaces* (you can jump directly to a symbol with `/`), or look up the symbol in the Kconfig symbol reference.

The Initial Configuration

The initial configuration for an application comes from merging configuration settings from three sources:

1. A `BOARD`-specific configuration file stored in `boards/<architecture>/<BOARD>/<BOARD>_defconfig`
2. Any CMake cache entries prefix with `CONFIG_`
3. The application configuration

The application configuration can come from the sources below. By default, `prj.conf` is used.

1. If `CONF_FILE` is set, the configuration file(s) specified in it are merged and used as the application configuration. `CONF_FILE` can be set in various ways:
    1. In `CMakeLists.txt`, before calling `find_package(Zephyr)`
    2. By passing `-DCONF_FILE=<conf file(s)>`, either directly or via `west`
    3. From the CMake variable cache
2. Otherwise if `CONF_FILE` is set, and a single configuration file of the form `prj_<build>.conf` is used, then if file `boards/<BOARD>_<build>.conf` exists in same folder as file `prj_<build>.conf`, the result of merging `prj_<build>.conf` and `boards/<BOARD>_<build>.conf` is used.
3. Otherwise, `prj_<BOARD>.conf` is used if it exists in the application directory.
4. Otherwise, if `boards/<BOARD>.conf` exists in the application directory, the result of merging it with `prj.conf` is used.

5. Otherwise, if board revisions are used and `boards/<BOARD>_<revision>.conf` exists in the application directory, the result of merging it with `prj.conf` and `boards/<BOARD>.conf` is used.
6. Otherwise, `prj.conf` is used if it exists in the application directory

If a symbol is assigned both in `<BOARD>_defconfig` and in the application configuration, the value set in the application configuration takes precedence.

The merged configuration is saved to `zephyr/.config` in the build directory.

As long as `zephyr/.config` exists and is up-to-date (is newer than any `BOARD` and application configuration files), it will be used in preference to producing a new merged configuration. `zephyr/.config` is also the configuration that gets modified when making changes in the *interactive configuration interfaces*.

Configuring invisible Kconfig symbols

When making changes to the default configuration for a board, you might have to configure invisible symbols. This is done in `boards/<architecture>/<BOARD>/Kconfig.defconfig`, which is a regular `Kconfig` file.

Note: Assignments in `.config` files have no effect on invisible symbols, so this scheme is not just an organizational issue.

Assigning values in `Kconfig.defconfig` relies on defining a Kconfig symbol in multiple locations. As an example, say we want to set `FOO_WIDTH` below to 32:

```
config FOO_WIDTH
    int
```

To do this, we extend the definition of `FOO_WIDTH` as follows, in `Kconfig.defconfig`:

```
if BOARD_MY_BOARD

config FOO_WIDTH
    default 32

endif
```

Note: Since the type of the symbol (`int`) has already been given at the first definition location, it does not need to be repeated here. Only giving the type once at the "base" definition of the symbol is a good idea for reasons explained in *Common Kconfig shorthands*.

`default` values in `Kconfig.defconfig` files have priority over `default` values given on the "base" definition of a symbol. Internally, this is implemented by including the `Kconfig.defconfig` files first. Kconfig uses the first `default` with a satisfied condition, where an empty condition corresponds to `if y` (is always satisfied).

Note that conditions from surrounding top-level `if`s are propagated to symbol properties, so the above `default` is equivalent to `default 32 if BOARD_MY_BOARD`.

Warning: When defining a symbol in multiple locations, dependencies are ORed together rather than ANDed together. It is not possible to make the dependencies of a symbol more restrictive by defining it in multiple locations.

For example, the direct dependencies of the symbol below becomes `DEP1 || DEP2`:

```
config FOO
  ...
  depends on DEP1

config FOO
  ...
  depends on DEP2
```

When making changes to `Kconfig.defconfig` files, always check the symbol's direct dependencies in one of the *interactive configuration interfaces* afterwards. It is often necessary to repeat dependencies from the base definition of the symbol to avoid weakening a symbol's dependencies.

Motivation for Kconfig.defconfig files One motivation for this configuration scheme is to avoid making fixed `BOARD`-specific settings configurable in the interactive configuration interfaces. If all board configuration were done via `<BOARD>_defconfig`, all symbols would have to be visible, as values given in `<BOARD>_defconfig` have no effect on invisible symbols.

Having fixed settings be user-configurable would clutter up the configuration interfaces and make them harder to understand, and would make it easier to accidentally create broken configurations.

When dealing with fixed board-specific settings, also consider whether they should be handled via *devicetree* instead.

Configuring choices There are two ways to configure a Kconfig `choice`:

1. By setting one of the choice symbols to `y` in a configuration file.

   Setting one choice symbol to `y` automatically gives all other choice symbols the value `n`.

   If multiple choice symbols are set to `y`, only the last one set to `y` will be honored (the rest will get the value `n`). This allows a choice selection from a board `defconfig` file to be overridden from an application `prj.conf` file.

2. By changing the `default` of the choice in `Kconfig.defconfig`.

   As with symbols, changing the default for a choice is done by defining the choice in multiple locations. For this to work, the choice must have a name.

   As an example, assume that a choice has the following base definition (here, the name of the choice is `FOO`):

   ```
   choice FOO
       bool "Foo choice"
       default B

   config A
       bool "A"

   config B
       bool "B"

   endchoice
   ```

   To change the default symbol of `FOO` to `A`, you would add the following definition to `Kconfig.defconfig`:

   ```
   choice FOO
       default A
   endchoice
   ```

The `Kconfig.defconfig` method should be used when the dependencies of the choice might not be satisfied. In that case, you're setting the default selection whenever the user makes the choice visible.

More Kconfig resources The *Kconfig - Tips and Best Practices* page has some tips for writing Kconfig files.

The kconfiglib.py docstring docstring (at the top of the file) goes over how symbol values are calculated in detail.

5.2.3 Kconfig - Tips and Best Practices

This page covers some Kconfig best practices and explains some Kconfig behaviors and features that might be cryptic or that are easily overlooked.

Note: The official Kconfig documentation is kconfig-language.rst and kconfig-macro-language.rst.

What to turn into Kconfig options

When deciding whether something belongs in Kconfig, it helps to distinguish between symbols that have prompts and symbols that don't.

If a symbol has a prompt (e.g. `bool "Enable foo"`), then the user can change the symbol's value in the `menuconfig` or `guiconfig` interface (see *Interactive Kconfig interfaces*), or by manually editing configuration files. Conversely, a symbol without a prompt can never be changed directly by the user, not even by manually editing configuration files.

Only put a prompt on a symbol if it makes sense for the user to change its value.

Symbols without prompts are called *hidden* or *invisible* symbols, because they don't show up in `menuconfig` and `guiconfig`. Symbols that have prompts can also be invisible, when their dependencies are not satisfied.

Symbols without prompts can't be configured directly by the user (they derive their value from other symbols), so less restrictions apply to them. If some derived setting is easier to calculate in Kconfig than e.g. during the build, then do it in Kconfig, but keep the distinction between symbols with and without prompts in mind.

See the *optional prompts* section for a way to deal with settings that are fixed on some machines and configurable on other machines.

What not to turn into Kconfig options

In Zephyr, Kconfig configuration is done after selecting a target board. In general, it does not make sense to use Kconfig for a value that corresponds to a fixed machine-specific setting. Usually, such settings should be handled via *devicetree* instead.

In particular, avoid adding new Kconfig options of the following types:

Options enabling individual devices Existing examples like `CONFIG_I2C_0` and `CONFIG_I2C_1` were introduced before Zephyr supported devicetree, and new cases are discouraged. See *Write device drivers using devicetree APIs* for details on how to do this with devicetree instead.

Options that specify a device in the system by name For example, if you are writing an I2C device driver, avoid creating an option named `MY_DEVICE_I2C_BUS_NAME` for specifying the bus node your device is controlled by. See *Device drivers that depend on other devices* for alternatives.

Similarly, if your application depends on a hardware-specific PWM device to control an RGB LED, avoid creating an option like `MY_PWM_DEVICE_NAME`. See *Applications that depend on board-specific devices* for alternatives.

Options that specify fixed hardware configuration For example, avoid Kconfig options specifying a GPIO pin.

An alternative applicable to device drivers is to define a GPIO specifier with type phandle-array in the device binding, and using the *GPIO* devicetree API from C. Similar advice applies to other cases where

devicetree.h provides *Hardware specific APIs* for referring to other nodes in the system. Search the source code for drivers using these APIs for examples.

An application-specific devicetree *binding* to identify board specific properties may be appropriate. See tests/drivers/gpio/gpio_basic_api for an example.

For applications, see blinky-sample for a devicetree-based alternative.

`select` statements

The `select` statement is used to force one symbol to `y` whenever another symbol is `y`. For example, the following code forces `CONSOLE` to `y` whenever `USB_CONSOLE` is `y`:

```
config CONSOLE
     bool "Console support"

...

config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
```

This section covers some pitfalls and good uses for `select`.

`select` pitfalls `select` might seem like a generally useful feature at first, but can cause configuration issues if overused.

For example, say that a new dependency is added to the `CONSOLE` symbol above, by a developer who is unaware of the `USB_CONSOLE` symbol (or simply forgot about it):

```
config CONSOLE
     bool "Console support"
     depends on STRING_ROUTINES
```

Enabling `USB_CONSOLE` now forces `CONSOLE` to `y`, even if `STRING_ROUTINES` is `n`.

To fix the problem, the `STRING_ROUTINES` dependency needs to be added to `USB_CONSOLE` as well:

```
config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
     depends on STRING_ROUTINES

...

config STRING_ROUTINES
     bool "Include string routines"
```

More insidious cases with dependencies inherited from `if` and `menu` statements are common.

An alternative attempt to solve the issue might be to turn the `depends on` into another `select`:

```
config CONSOLE
     bool "Console support"
     select STRING_ROUTINES

...

config USB_CONSOLE
```

(continues on next page)

(continued from previous page)

```
    bool "USB console support"
    select CONSOLE
```

In practice, this often amplifies the problem, because any dependencies added to `STRING_ROUTINES` now need to be copied to both `CONSOLE` and `USB_CONSOLE`.

In general, whenever the dependencies of a symbol are updated, the dependencies of all symbols that (directly or indirectly) select it have to be updated as well. This is very often overlooked in practice, even for the simplest case above.

Chains of symbols selecting each other should be avoided in particular, except for simple helper symbols, as covered below in *Using select for helper symbols*.

Liberal use of `select` also tends to make Kconfig files harder to read, both due to the extra dependencies and due to the non-local nature of `select`, which hides ways in which a symbol might get enabled.

Alternatives to `select` For the example in the previous section, a better solution is usually to turn the `select` into a `depends on`:

```
config CONSOLE
    bool "Console support"

...

config USB_CONSOLE
    bool "USB console support"
    depends on CONSOLE
```

This makes it impossible to generate an invalid configuration, and means that dependencies only ever have to be updated in a single spot.

An objection to using `depends on` here might be that configuration files that enable `USB_CONSOLE` now also need to enable `CONSOLE`:

```
CONFIG_CONSOLE=y
CONFIG_USB_CONSOLE=y
```

This comes down to a trade-off, but if enabling `CONSOLE` is the norm, then a mitigation is to make `CONSOLE` default to `y`:

```
config CONSOLE
    bool "Console support"
    default y
```

This gives just a single assignment in configuration files:

```
CONFIG_USB_CONSOLE=y
```

Note that configuration files that do not want `CONSOLE` enabled now have to explicitly disable it:

```
CONFIG_CONSOLE=n
```

Using `select` for helper symbols A good and safe use of `select` is for setting "helper" symbols that capture some condition. Such helper symbols should preferably have no prompt or dependencies.

For example, a helper symbol for indicating that a particular CPU/SoC has an FPU could be defined as follows:

```
config CPU_HAS_FPU
     bool
     help
       If y, the CPU has an FPU

...

config SOC_FOO
     bool "FOO SoC"
     select CPU_HAS_FPU

...

config SOC_BAR
     bool "BAR SoC"
     select CPU_HAS_FPU
```

This makes it possible for other symbols to check for FPU support in a generic way, without having to look for particular architectures:

```
config FPU
     bool "Support floating point operations"
     depends on CPU_HAS_FPU
```

The alternative would be to have dependencies like the following, possibly duplicated in several spots:

```
config FPU
     bool "Support floating point operations"
     depends on SOC_FOO || SOC_BAR || ...
```

Invisible helper symbols can also be useful without `select`. For example, the following code defines a helper symbol that has the value `y` if the machine has some arbitrarily-defined "large" amount of memory:

```
config LARGE_MEM
     def_bool MEM_SIZE >= 64
```

Note: This is short for the following:

```
config LARGE_MEM
  bool
  default MEM_SIZE >= 64
```

`select` **recommendations** In summary, here are some recommended practices for `select`:

- Avoid selecting symbols with prompts or dependencies. Prefer `depends on`. If `depends on` causes annoying bloat in configuration files, consider adding a Kconfig default for the most common value.

  Rare exceptions might include cases where you're sure that the dependencies of the selecting and selected symbol will never drift out of sync, e.g. when dealing with two simple symbols defined close to one another within the same `if`.

  Common sense applies, but be aware that `select` often causes issues in practice. `depends on` is usually a cleaner and safer solution.

- Select simple helper symbols without prompts and dependencies however much you like. They're a great tool for simplifying Kconfig files.

(Lack of) conditional includes

`if` blocks add dependencies to each item within the `if`, as if `depends on` was used.

A common misunderstanding related to `if` is to think that the following code conditionally includes the file `Kconfig.other`:

```
if DEP
source "Kconfig.other"
endif
```

In reality, there are no conditional includes in Kconfig. `if` has no special meaning around a `source`.

Note: Conditional includes would be impossible to implement, because `if` conditions may contain (either directly or indirectly) forward references to symbols that haven't been defined yet.

Say that `Kconfig.other` above contains this definition:

```
config FOO
     bool "Support foo"
```

In this case, `FOO` will end up with this definition:

```
config FOO
     bool "Support foo"
     depends on DEP
```

Note that it is redundant to add `depends on DEP` to the definition of `FOO` in `Kconfig.other`, because the `DEP` dependency has already been added by `if DEP`.

In general, try to avoid adding redundant dependencies. They can make the structure of the Kconfig files harder to understand, and also make changes more error-prone, since it can be hard to spot that the same dependency is added twice.

"Stuck" symbols in menuconfig and guiconfig

There is a common subtle gotcha related to interdependent configuration symbols with prompts. Consider these symbols:

```
config FOO
     bool "Foo"

config STACK_SIZE
     hex "Stack size"
     default 0x200 if FOO
     default 0x100
```

Assume that the intention here is to use a larger stack whenever `FOO` is enabled, and that the configuration initially has `FOO` disabled. Also, remember that Zephyr creates an initial configuration in `zephyr/.config` in the build directory by merging configuration files (including e.g. `prj.conf`). This configuration file exists before `menuconfig` or `guiconfig` is run.

When first entering the configuration interface, the value of `STACK_SIZE` is 0x100, as expected. After enabling `FOO`, you might reasonably expect the value of `STACK_SIZE` to change to 0x200, but it stays as 0x100.

To understand what's going on, remember that `STACK_SIZE` has a prompt, meaning it is user-configurable, and consider that all Kconfig has to go on from the initial configuration is this:

```
CONFIG_STACK_SIZE=0x100
```

Since Kconfig can't know if the 0x100 value came from a `default` or was typed in by the user, it has to assume that it came from the user. Since `STACK_SIZE` is user-configurable, the value from the configuration file is respected, and any symbol defaults are ignored. This is why the value of `STACK_SIZE` appears to be "frozen" at 0x100 when toggling `FOO`.

The right fix depends on what the intention is. Here's some different scenarios with suggestions:

- If `STACK_SIZE` can always be derived automatically and does not need to be user-configurable, then just remove the prompt:

  ```
  config STACK_SIZE
     hex
     default 0x200 if FOO
     default 0x100
  ```

  Symbols without prompts ignore any value from the saved configuration.

- If `STACK_SIZE` should usually be user-configurable, but needs to be set to 0x200 when `FOO` is enabled, then disable its prompt when `FOO` is enabled, as described in *optional prompts*:

  ```
  config STACK_SIZE
     hex "Stack size" if !FOO
     default 0x200 if FOO
     default 0x100
  ```

- If `STACK_SIZE` should usually be derived automatically, but needs to be set to a custom value in rare circumstances, then add another option for making `STACK_SIZE` user-configurable:

  ```
  config CUSTOM_STACK_SIZE
     bool "Use a custom stack size"
     help
       Enable this if you need to use a custom stack size. When disabled, a
       suitable stack size is calculated automatically.

  config STACK_SIZE
     hex "Stack size" if CUSTOM_STACK_SIZE
     default 0x200 if FOO
     default 0x100
  ```

  As long as `CUSTOM_STACK_SIZE` is disabled, `STACK_SIZE` will ignore the value from the saved configuration.

It is a good idea to try out changes in the `menuconfig` or `guiconfig` interface, to make sure that things behave the way you expect. This is especially true when making moderately complex changes like these.

Assignments to promptless symbols in configuration files

Assignments to hidden (promptless, also called *invisible*) symbols in configuration files are always ignored. Hidden symbols get their value indirectly from other symbols, via e.g. `default` and `select`.

A common source of confusion is opening the output configuration file (`zephyr/.config`), seeing a bunch of assignments to hidden symbols, and assuming that those assignments must be respected when the configuration is read back in by Kconfig. In reality, all assignments to hidden symbols in `zephyr/.config` are ignored by Kconfig, like for other configuration files.

To understand why `zephyr/.config` still includes assignments to hidden symbols, it helps to realize that `zephyr/.config` serves two separate purposes:

1. It holds the saved configuration, and

2. it holds configuration output. `zephyr/.config` is parsed by the CMake files to let them query configuration settings, for example.

The assignments to hidden symbols in `zephyr/.config` are just configuration output. Kconfig itself ignores assignments to hidden symbols when calculating symbol values.

Note: A *minimal configuration*, which can be generated from within the *menuconfig and guiconfig interfaces*, could be considered closer to just a saved configuration, without the full configuration output.

`depends on` and `string/int/hex` symbols

`depends on` works not just for `bool` symbols, but also for `string`, `int`, and `hex` symbols (and for choices).

The Kconfig definitions below will hide the `FOO_DEVICE_FREQUENCY` symbol and disable any configuration output for it when `FOO_DEVICE` is disabled.

```
config FOO_DEVICE
     bool "Foo device"

config FOO_DEVICE_FREQUENCY
     int "Foo device frequency"
     depends on FOO_DEVICE
```

In general, it's a good idea to check that only relevant symbols are ever shown in the `menuconfig/guiconfig` interface. Having `FOO_DEVICE_FREQUENCY` show up when `FOO_DEVICE` is disabled (and possibly hidden) makes the relationship between the symbols harder to understand, even if code never looks at `FOO_DEVICE_FREQUENCY` when `FOO_DEVICE` is disabled.

`menuconfig` symbols

If the definition of a symbol `FOO` is immediately followed by other symbols that depend on `FOO`, then those symbols become children of `FOO`. If `FOO` is defined with `config FOO`, then the children are shown indented relative to `FOO`. Defining `FOO` with `menuconfig FOO` instead puts the children in a separate menu rooted at `FOO`.

`menuconfig` has no effect on evaluation. It's just a display option.

`menuconfig` can cut down on the number of menus and make the menu structure easier to navigate. For example, say you have the following definitions:

```
menu "Foo subsystem"

config FOO_SUBSYSTEM
     bool "Foo subsystem"

if FOO_SUBSYSTEM

config FOO_FEATURE_1
     bool "Foo feature 1"

config FOO_FEATURE_2
     bool "Foo feature 2"

config FOO_FREQUENCY
     int "Foo frequency"
```

(continues on next page)

(continued from previous page)

```
... lots of other FOO-related symbols

endif # FOO_SUBSYSTEM

endmenu
```

In this case, it's probably better to get rid of the `menu` and turn `FOO_SUBSYSTEM` into a `menuconfig` symbol:

```
menuconfig FOO_SUBSYSTEM
     bool "Foo subsystem"

if FOO_SUBSYSTEM

config FOO_FEATURE_1
     bool "Foo feature 1"

config FOO_FEATURE_2
     bool "Foo feature 2"

config FOO_FREQUENCY
     int "Foo frequency"

... lots of other FOO-related symbols

endif # FOO_SUBSYSTEM
```

In the `menuconfig` interface, this will be displayed as follows:

```
[*] Foo subsystem  --->
```

Note that making a symbol without children a `menuconfig` is meaningless. It should be avoided, because it looks identical to a symbol with all children invisible:

```
[*] I have no children  ----
[*] All my children are invisible  ----
```

Checking changes in menuconfig/guiconfig

When adding new symbols or making other changes to Kconfig files, it is a good idea to look up the symbols in *menuconfig or guiconfig* afterwards. To get to a symbol quickly, use the jump-to feature (press `/`).

Here are some things to check:

- Are the symbols placed in a good spot? Check that they appear in a menu where they make sense, close to related symbols.

  If one symbol depends on another, then it's often a good idea to place it right after the symbol it depends on. It will then be shown indented relative to the symbol it depends on in the `menuconfig` interface, and in a separate menu rooted at the symbol in `guiconfig`. This also works if several symbols are placed after the symbol they depend on.

- Is it easy to guess what the symbols do from their prompts?
- If many symbols are added, do all combinations of values they can be set to make sense?

  For example, if two symbols `FOO_SUPPORT` and `NO_FOO_SUPPORT` are added, and both can be enabled at the same time, then that makes a nonsensical configuration. In this case, it's probably better to have a single `FOO_SUPPORT` symbol.

- Are there any duplicated dependencies?

  This can be checked by selecting a symbol and pressing `?` to view the symbol information. If there are duplicated dependencies, then use the `Included via ...` path shown in the symbol information to figure out where they come from.

Checking changes with `scripts/kconfig/lint.py`

After you make Kconfig changes, you can use the scripts/kconfig/lint.py script to check for some potential issues, like unused symbols and symbols that are impossible to enable. Use `--help` to see available options.

Some checks are necessarily a bit heuristic, so a symbol being flagged by a check does not necessarily mean there's a problem. If a check returns a false positive e.g. due to token pasting in C (`CONFIG_FOO_##index##_BAR`), just ignore it.

When investigating an unknown symbol `FOO_BAR`, it is a good idea to run `git grep FOO_BAR` to look for references. It is also a good idea to search for some components of the symbol name with e.g. `git grep FOO` and `git grep BAR`, as it can help uncover token pasting.

Style recommendations and shorthands

This section gives some style recommendations and explains some common Kconfig shorthands.

Factoring out common dependencies If a sequence of symbols/choices share a common dependency, the dependency can be factored out with an `if`.

As an example, consider the following code:

```
config FOO
     bool "Foo"
     depends on DEP

config BAR
     bool "Bar"
     depends on DEP

choice
     prompt "Choice"
     depends on DEP

config BAZ
     bool "Baz"

config QAZ
     bool "Qaz"

endchoice
```

Here, the `DEP` dependency can be factored out like this:

```
if DEP

config FOO
     bool "Foo"

config BAR
```

(continues on next page)

(continued from previous page)

```
     bool "Bar"

choice
     prompt "Choice"

config BAZ
     bool "Baz"

config QAZ
     bool "Qaz"

endchoice

endif # DEP
```

Note: Internally, the second version of the code is transformed into the first.

If a sequence of symbols/choices with shared dependencies are all in the same menu, the dependency can be put on the menu itself:

```
menu "Foo features"
     depends on FOO_SUPPORT

config FOO_FEATURE_1
     bool "Foo feature 1"

config FOO_FEATURE_2
     bool "Foo feature 2"

endmenu
```

If `FOO_SUPPORT` is `n`, the entire menu disappears.

Redundant defaults `bool` symbols implicitly default to `n`, and `string` symbols implicitly default to the empty string. Therefore, `default n` and `default ""` are (almost) always redundant.

The recommended style in Zephyr is to skip redundant defaults for `bool` and `string` symbols. That also generates clearer documentation: (*Implicitly defaults to n* instead of *n if <dependencies, possibly inherited>*).

Note: The one case where `default n`/`default ""` is not redundant is when defining a symbol in multiple locations and wanting to override e.g. a `default y` on a later definition.

Defaults *should* always be given for `int` and `hex` symbols, however, as they implicitly default to the empty string. This is partly for compatibility with the C Kconfig tools, though an implicit 0 default might be less likely to be what was intended compared to other symbol types as well.

Common Kconfig shorthands Kconfig has two shorthands that deal with prompts and defaults.

- `<type> "prompt"` is a shorthand for giving a symbol/choice a type and a prompt at the same time. These two definitions are equal:

  ```
  config FOO
     bool "foo"
  ```

```
config FOO
   bool
   prompt "foo"
```

The first style, with the shorthand, is preferred in Zephyr.

- `def_<type> <value>` is a shorthand for giving a type and a value at the same time. These two definitions are equal:

```
config FOO
   def_bool BAR && BAZ
```

```
config FOO
   bool
   default BAR && BAZ
```

Using both the `<type> "prompt"` and the `def_<type> <value>` shorthand in the same definition is redundant, since it gives the type twice.

The `def_<type> <value>` shorthand is generally only useful for symbols without prompts, and somewhat obscure.

Note: For a symbol defined in multiple locations (e.g., in a `Kconfig.defconfig` file in Zephyr), it is best to only give the symbol type for the "base" definition of the symbol, and to use `default` (instead of `def_<type> value`) for the remaining definitions. That way, if the base definition of the symbol is removed, the symbol ends up without a type, which generates a warning that points to the other definitions. That makes the extra definitions easier to discover and remove.

Prompt strings For a Kconfig symbol that enables a driver/subsystem FOO, consider having just "Foo" as the prompt, instead of "Enable Foo support" or the like. It will usually be clear in the context of an option that can be toggled on/off, and makes things consistent.

Header comments and other nits A few formatting nits, to help keep things consistent:

- Use this format for any header comments at the top of `Kconfig` files:

```
# <Overview of symbols defined in the file, preferably in plain English>
(Blank line)
# Copyright (c) 2019 ...
# SPDX-License-Identifier: <License>
(Blank line)
(Kconfig definitions)
```

- Format comments as `# Comment` rather than `#Comment`
- Put a blank line before/after each top-level `if` and `endif`
- Use a single tab for each indentation
- Indent help text with two extra spaces

Lesser-known/used Kconfig features

This section lists some more obscure Kconfig behaviors and features that might still come in handy.

The `imply` statement The `imply` statement is similar to `select`, but respects dependencies and doesn't force a value. For example, the following code could be used to enable USB keyboard support by default on the FOO SoC, while still allowing the user to turn it off:

```
config SOC_FOO
     bool "FOO SoC"
     imply USB_KEYBOARD

...

config USB_KEYBOARD
     bool "USB keyboard support"
```

`imply` acts like a suggestion, whereas `select` forces a value.

Optional prompts A condition can be put on a symbol's prompt to make it optionally configurable by the user. For example, a value `MASK` that's hardcoded to 0xFF on some boards and configurable on others could be expressed as follows:

```
config MASK
     hex "Bitmask" if HAS_CONFIGURABLE_MASK
     default 0xFF
```

Note: This is short for the following:

```
config MASK
  hex
  prompt "Bitmask" if HAS_CONFIGURABLE_MASK
  default 0xFF
```

The `HAS_CONFIGURABLE_MASK` helper symbol would get selected by boards to indicate that `MASK` is configurable. When `MASK` is configurable, it will also default to 0xFF.

Optional choices Defining a choice with the `optional` keyword allows the whole choice to be toggled off to select none of the symbols:

```
choice
     prompt "Use legacy protocol"
     optional

config LEGACY_PROTOCOL_1
     bool "Legacy protocol 1"

config LEGACY_PROTOCOL_2
     bool "Legacy protocol 2"

endchoice
```

In the `menuconfig` interface, this will be displayed e.g. as `[*] Use legacy protocol (Legacy protocol 1) --->`, where the choice can be toggled off to enable neither of the symbols.

`visible if` conditions Putting a `visible if` condition on a menu hides the menu and all the symbols within it, while still allowing symbol default values to kick in.

As a motivating example, consider the following code:

```
menu "Foo subsystem"
     depends on HAS_CONFIGURABLE_FOO

config FOO_SETTING_1
     int "Foo setting 1"
     default 1

config FOO_SETTING_2
     int "Foo setting 2"
     default 2

endmenu
```

When `HAS_CONFIGURABLE_FOO` is `n`, no configuration output is generated for `FOO_SETTING_1` and `FOO_SETTING_2`, as the code above is logically equivalent to the following code:

```
config FOO_SETTING_1
     int "Foo setting 1"
     default 1
     depends on HAS_CONFIGURABLE_FOO

config FOO_SETTING_2
     int "Foo setting 2"
     default 2
     depends on HAS_CONFIGURABLE_FOO
```

If we want the symbols to still get their default values even when `HAS_CONFIGURABLE_FOO` is `n`, but not be configurable by the user, then we can use `visible if` instead:

```
menu "Foo subsystem"
     visible if HAS_CONFIGURABLE_FOO

config FOO_SETTING_1
     int "Foo setting 1"
     default 1

config FOO_SETTING_2
     int "Foo setting 2"
     default 2

endmenu
```

This is logically equivalent to the following:

```
config FOO_SETTING_1
     int "Foo setting 1" if HAS_CONFIGURABLE_FOO
     default 1

config FOO_SETTING_2
     int "Foo setting 2" if HAS_CONFIGURABLE_FOO
     default 2
```

Note: See the *optional prompts* section for the meaning of the conditions on the prompts.

When `HAS_CONFIGURABLE` is `n`, we now get the following configuration output for the symbols, instead of no output:

```
...
CONFIG_FOO_SETTING_1=1
CONFIG_FOO_SETTING_2=2
...
```

Other resources

The *Intro to symbol values* section in the Kconfiglib docstring goes over how symbols values are calculated in more detail.

5.2.4 Custom Kconfig Preprocessor Functions

Kconfiglib supports custom Kconfig preprocessor functions written in Python. These functions are defined in scripts/kconfig/kconfigfunctions.py.

Note: The official Kconfig preprocessor documentation can be found here.

Most of the custom preprocessor functions are used to get devicetree information into Kconfig. For example, the default value of a Kconfig symbol can be fetched from a devicetree `reg` property.

Devicetree-related Functions

The functions listed below are used to get devicetree information into Kconfig. See the Python docstrings in scripts/kconfig/kconfigfunctions.py for detailed documentation.

The `*_int` version of each function returns the value as a decimal integer, while the `*_hex` version returns a hexadecimal value starting with `0x`.

```
$(dt_chosen_reg_addr_int,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_addr_hex,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_size_int,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_size_hex,<property in /chosen>[,<index>,<unit>])
$(dt_node_reg_addr_int,<node path>[,<index>,<unit>])
$(dt_node_reg_addr_hex,<node path>[,<index>,<unit>])
$(dt_node_reg_size_int,<node path>[,<index>,<unit>])
$(dt_node_reg_size_hex,<node path>[,<index>,<unit>])
$(dt_compat_enabled,<compatible string>)
$(dt_chosen_enabled,<property in /chosen>)
$(dt_node_has_bool_prop,<node path>,<prop>)
$(dt_node_has_prop,<node path>,<prop>)
```

Example Usage Assume that the devicetree for some board looks like this:

```
{
     soc {
             #address-cells = <1>;
             #size-cells = <1>;

             spi0: spi@10014000 {
                     compatible = "sifive,spi0";
                     reg = <0x10014000 0x1000 0x20010000 0x3c0900>;
                     reg-names = "control", "mem";
```

(continues on next page)

(continued from previous page)

```
                    ...
            };
};
```

The second entry in `reg` in `spi@1001400` (`<0x20010000 0x3c0900>`) corresponds to `mem`, and has the address `0x20010000`. This address can be inserted into Kconfig as follows:

```
config FLASH_BASE_ADDRESS
     default $(dt_node_reg_addr_hex,/soc/spi@1001400,1)
```

After preprocessor expansion, this turns into the definition below:

```
config FLASH_BASE_ADDRESS
     default 0x20010000
```

5.2.5 Kconfig extensions

Zephyr uses the Kconfiglib implementation of Kconfig, which includes some Kconfig extensions:

- Environment variables in `source` statements are expanded directly, meaning no "bounce" symbols with `option env="ENV_VAR"` need to be defined.

  Note: `option env` has been removed from the C tools as of Linux 4.18 as well.

  The recommended syntax for referencing environment variables is `$(FOO)` rather than `$FOO`. This uses the new Kconfig preprocessor. The `$FOO` syntax for expanding environment variables is only supported for backwards compatibility.

- The `source` statement supports glob patterns and includes each matching file. A pattern is required to match at least one file.

  Consider the following example:

  ```
  source "foo/bar/*/Kconfig"
  ```

  If the pattern `foo/bar/*/Kconfig` matches the files `foo/bar/baz/Kconfig` and `foo/bar/qaz/Kconfig`, the statement above is equivalent to the following two `source` statements:

  ```
  source "foo/bar/baz/Kconfig"
  source "foo/bar/qaz/Kconfig"
  ```

  If no files match the pattern, an error is generated.

  The wildcard patterns accepted are the same as for the Python glob module.

  For cases where it's okay for a pattern to match no files (or for a plain filename to not exist), a separate `osource` (*optional source*) statement is available. `osource` is a no-op if no file matches.

  Note: `source` and `osource` are analogous to `include` and `-include` in Make.

- An `rsource` statement is available for including files specified with a relative path. The path is relative to the directory of the `Kconfig` file that contains the `rsource` statement.

  As an example, assume that `foo/Kconfig` is the top-level `Kconfig` file, and that `foo/bar/Kconfig` has the following statements:

```
source "qaz/Kconfig1"
rsource "qaz/Kconfig2"
```

This will include the two files `foo/qaz/Kconfig1` and `foo/bar/qaz/Kconfig2`.

`rsource` can be used to create `Kconfig` "subtrees" that can be moved around freely.

`rsource` also supports glob patterns.

A drawback of `rsource` is that it can make it harder to figure out where a file gets included, so only use it if you need it.

- An `orsource` statement is available that combines `osource` and `rsource`.

  For example, the following statement will include `Kconfig1` and `Kconfig2` from the current directory (if they exist):

```
orsource "Kconfig[12]"
```

- `def_int`, `def_hex`, and `def_string` keywords are available, analogous to `def_bool`. These set the type and add a `default` at the same time.

Users interested in optimizing their configuraion for security should refer to the Zephyr Security Guide's section on the *Hardening Tool*.

Chapter 6

Application Development

Note: In this document, we'll assume your **application directory** is `<home>/app`, and that its **build directory** is `<home>/app/build`. (These terms are defined in the following Overview.) On Linux/macOS, <home> is equivalent to `~`, whereas on Windows it's `%userprofile%`.

6.1 Overview

Zephyr's build system is based on CMake.

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

Zephyr's base directory hosts Zephyr's own source code, its kernel configuration options, and its build definitions.

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

An application in its simplest form has the following contents:

```
<home>/app
├── CMakeLists.txt
├── prj.conf
└── src
    └── main.c
```

These contents are:

- **CMakeLists.txt**: This file tells the build system where to find the other application files, and links the application directory with Zephyr's CMake build system. This link provides features supported by Zephyr's build system, such as board-specific kernel configuration files, the ability to run and debug compiled binaries on real or emulated hardware, and more.
- **Kernel configuration files**: An application typically provides a Kconfig configuration file (usually called `prj.conf`) that specifies application-specific values for one or more kernel configuration options. These application settings are merged with board-specific settings to produce a kernel configuration.

  See *Kconfig Configuration* below for more information.

- **Application source code files**: An application typically provides one or more application-specific files, written in C or assembly language. These files are usually located in a sub-directory called `src`.

Once an application has been defined, you can use CMake to create project files for building it from a directory where you want to host these files. This is known as the **build directory**. Application build artifacts are always generated in a build directory; Zephyr does not support "in-tree" builds.

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

6.2 Source Tree Structure

Understanding the Zephyr source tree can be helpful in locating the code associated with a particular Zephyr feature.

At the top of the tree there are several files that are of importance:

`CMakeLists.txt` The top-level file for the CMake build system, containing a lot of the logic required to build Zephyr.

`Kconfig` The top-level Kconfig file, which refers to the file `Kconfig.zephyr` also found at the top-level directory.

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

`west.yml` The *West (Zephyr's meta-tool)* manifest, listing the external repositories managed by the west command-line tool.

The Zephyr source tree also contains the following top-level directories, each of which may have one or more additional levels of subdirectories which are not described here.

`arch` Architecture-specific kernel and system-on-chip (SoC) code. Each supported architecture (for example, x86 and ARM) has its own subdirectory, which contains additional subdirectories for the following areas:

- architecture-specific kernel source files
- architecture-specific kernel include files for private APIs

`soc` SoC related code and configuration files.

`boards` Board related code and configuration files.

`doc` Zephyr technical documentation source files and tools used to generate the https://docs.zephyrproject.org web content.

`drivers` Device driver code.

`dts` *devicetree* source files used to describe non-discoverable board-specific hardware details.

`include` Include files for all public APIs, except those defined under `lib`.

`kernel` Architecture-independent kernel code.

`lib` Library code, including the minimal standard C library.

`misc` Miscellaneous code that doesn't belong to any of the other top-level directories.

`samples` Sample applications that demonstrate the use of Zephyr features.

`scripts` Various programs and other files used to build and test Zephyr applications.

`cmake` Additional build scripts needed to build Zephyr.

`subsys` Subsystems of Zephyr, including:

- USB device stack code.

- Networking code, including the Bluetooth stack and networking stacks.
- File system code.
- Bluetooth host and controller

`tests` Test code and benchmarks for Zephyr features.

`share` Additional architecture independent data. Currently containing Zephyr CMake package.

6.3 Example standalone application

A reference standalone application contained in its own Git repository can be found in the Example Application repository. It can be used as a reference on how to structure out-of-tree, Zephyr-based applications using the *T2 star topology*. It also demonstrates the out-of-tree use of features commonly used in applications such as:

- Custom boards
- Custom devicetree bindings
- Custom drivers
- Continuous Integration (CI) setup

6.4 Creating an Application

Follow these steps to create a new application directory. (Refer to the Example Application repository for a reference standalone application in its own Git repository or to samples-and-demos for existing applications provided as part of Zephyr.)

1. Create an application directory on your workstation computer, outside of the Zephyr base directory. Usually you'll want to create it somewhere under your user's home directory.

   For example, in a Unix shell or Windows `cmd.exe` prompt, navigate to where you want to create your application, then enter:

   ```
   mkdir app
   ```

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

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

   Continuing the previous example, enter:

   ```
   cd app
   mkdir src
   ```

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

```
# Find Zephyr. This also loads Zephyr's build system.
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr)
project(my_zephyr_app)

# Add your source file to the "app" target. This must come after
# find_package(Zephyr) which defines the target.
target_sources(app PRIVATE src/main.c)
```

`find_package(Zephyr)` sets the minimum CMake version and pulls in the Zephyr build system, which creates a CMake target named `app` (see *Zephyr CMake Package*). Adding sources to this target is how you include them in the build.

Note: `cmake_minimum_required()` is also invoked by the Zephyr package. The most recent of the two versions will be enforced by CMake.

---

5. Set Kconfig configuration options. See *Kconfig Configuration*.
6. Configure any devicetree overlays needed by your application. See *Set devicetree overlays*.

Note: `include($ENV{ZEPHYR_BASE}/cmake/app/boilerplate.cmake NO_POLICY_SCOPE)` is still supported for backward compatibility with older applications. Including `boilerplate.cmake` directly in the sample still requires to run `source zephyr-env.sh` or execute `zephyr-env.cmd` before building the application.

6.5 Setting Variables

6.5.1 Option 1: Just Once

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

```
# Linux and macOS
export MY_VARIABLE=foo

# Windows
set MY_VARIABLE=foo
```

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

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

6.5.2 Option 2: In all Terminals

macOS and Linux:

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

Windows:

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

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

```
setx MY_VARIABLE foo
```

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

```
choco install rapidee
```

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

6.5.3 Option 3: Using `zephyrrc` files

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

macOS and Linux:

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

```
export MY_VARIABLE=foo
```

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

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

Windows:

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

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

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

These scripts:

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

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

6.5.4 Option 4: Using Zephyr Build Configuration CMake package

Choose this option if you want to make those variable settings shared among all users of your project.

Using a *Zephyr Build Configuration CMake package* allows you to commit the shared settings into the repository, so that all users can share them.

It also removes the need for running `source zephyr-env.sh` or `zephyr-env.cmd` when opening a new terminal.

6.6 Important Build System Variables

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

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

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

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

  See *The Initial Configuration* for more information.
- `OVERLAY_CONFIG`: Additional Kconfig configuration fragment files. Multiple filenames can be separated with either spaces or semicolons. This can be useful in order to leave `CONF_FILE` at its default value, but "mix in" some additional configuration options.
- `DTC_OVERLAY_FILE`: One or more devicetree overlay files to use. Multiple files can be separated with semicolons. See *Set devicetree overlays* for examples and *Introduction to devicetree* for information about devicetree and Zephyr.
- `ZEPHYR_MODULES`: A CMake list containing absolute paths of additional directories with source code, Kconfig, etc. that should be used in the application build. See *Modules (External projects)* for details.

6.7 Application CMakeLists.txt

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

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

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

   ```
   set(BOARD qemu_x86)
   ```

   Refer to boards for more information on available boards.

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

   - Any previously used value as determined by the CMake cache takes highest precedence. This ensures you don't try to run a build with a different `BOARD` value than you set during the build configuration step.

- Any value given on the CMake command line (directly or indirectly via `west build`) using `-DBOARD=YOUR_BOARD` will be checked for and used next.
- If an *environment variable* `BOARD` is set, its value will then be used.
- Finally, if you set `BOARD` in your application `CMakeLists.txt` as described in this step, this value will be used.

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

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

   See *The Initial Configuration* for more information.

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

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

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

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

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

   Structure your `Kconfig` file like this:

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

mainmenu "Your Application Name"

# Your application configuration options go here

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

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

   See *Kconfig extensions* for more information.

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

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

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

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

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

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

Below is a simple example `CMakeList.txt`:

```
set(BOARD qemu_x86)

find_package(Zephyr)
project(my_zephyr_app)

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

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

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

6.8 CMakeCache.txt

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

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

6.9 Application Configuration

6.9.1 Kconfig Configuration

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

```
CONFIG_CPLUSPLUS=y
```

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

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

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

6.9.2 Devicetree Overlays

See *Set devicetree overlays*.

6.10 Application-Specific Code

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

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

6.10.1 Third-party Library Code

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

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

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

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

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

6.11 Building an Application

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

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

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

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

Using west:

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

Using CMake and ninja:

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

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

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

Using west:

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

Using CMake directly:

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

# Now run ninja on the generated build system:
make -C build
```

6.11.1 Basics

Note: In the below example, `west` is used outside of a west workspace. For this to work, you must set the `ZEPHYR_BASE` environment variable to the path of your zephyr git repository, using one of the methods on the *Environment Variables* page.

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

   Using west:

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

   Using CMake and ninja:

   ```
   mkdir build && cd build

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

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

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

   Using west:

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

Using CMake and ninja:

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

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

6.11.2 Build Directory Contents

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

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

The most notable files in the build directory are:

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

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

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

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

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

6.11.3 Rebuilding an Application

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

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

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

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

   ```
   west build -t clean
   ```

   or

   ```
   ninja clean
   ```

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

   ```
   west build -t pristine
   ```

   or

   ```
   ninja pristine
   ```

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

6.11.4 Building for a board revision

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

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

Using west:

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

Using CMake and ninja:

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

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

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

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

6.12 Run an Application

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

6.12.1 Running on a Board

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

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

   ```
   west flash
   ```

   or

   ```
   ninja flash
   ```

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

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

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

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

6.12.2 Running in an Emulator

The kernel has built-in emulator support for QEMU (on Linux/macOS only, this is not yet supported on Windows). It allows you to run and test an application virtually, before (or in lieu of) loading and running it on actual target hardware. Follow these instructions to run an application via QEMU:

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

   For example, you could set `BOARD` to:

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

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

   ```
   west build -t run
   ```

   or

   ```
   ninja run
   ```

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

   The application stops running and the terminal console prompt redisplays.

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

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

6.13 Application Debugging

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

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

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

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

We will use the standard 1234 TCP port to open a GDB (GNU Debugger) server instance. This port number can be changed for a port that best suits the development environment.

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

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

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

The options used above have the following meaning:

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

To debug with QEMU and to start a GDB server and wait for a remote connect, run either of the following inside the build directory of an application:

```
ninja debugserver
```

The build system will start a QEMU instance with the CPU halted at startup and with a GDB server instance listening at the TCP port 1234.

Using a local GDB configuration `.gdbinit` can help initialize your GDB instance on every run. In this example, the initialization file points to the GDB server instance. It configures a connection to a remote target at the local host on the TCP port 1234. The initialization sets the kernel's root directory as a reference.

The `.gdbinit` file contains the following lines:

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

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

Execute the application to debug from the same directory that you chose for the `gdbinit` file. The command can include the `--tui` option to enable the use of a terminal user interface. The following commands connects to the GDB server using `gdb`. The command loads the symbol table from the elf binary file. In this example, the elf binary file name corresponds to `zephyr.elf` file:

```
..../path/to/gdb --tui zephyr.elf
```

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

If you are not using a .gdbinit file, issue the following command inside GDB to connect to the remote GDB server on port 1234:

```
(gdb) target remote localhost:1234
```

Finally, the command below connects to the GDB server using the Data Displayer Debugger (`ddd`). The command loads the symbol table from the elf binary file, in this instance, the `zephyr.elf` file.

The DDD (Data Displayer Debugger) may not be installed in your development system by default. Follow your system instructions to install it. For example, use `sudo apt-get install ddd` on an Ubuntu system.

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

Both commands execute the GDB (GNU Debugger). The command name might change depending on the toolchain you are using and your cross-development tools.

6.14 Custom Board, Devicetree and SOC Definitions

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

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

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

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

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

```
.
├── boards
│   └── x86
│       └── my_custom_board
│           ├── doc
│           │   └── img
│           └── support
└── src
```

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

6.14.1 Boards

Use the proper architecture folder name (e.g., `x86`, `arm`, etc.) under `boards` for `my_custom_board`. (See boards for a list of board architectures.)

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

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

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

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

Using west:

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

Using CMake and ninja:

```
cmake -B build -GNinja -DBOARD=<board name> -DBOARD_ROOT=<path to boards> .
ninja -C build
```

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

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

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

6.14.2 SOC Definitions

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

```
soc
└── arm
    └── st_stm32
            ├── common
            └── stm3210
```

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

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

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

```
soc
└── arm
    └── st_stm32
            ├── Kconfig
            ├── Kconfig.soc
            └── Kconfig.defconfig
```

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

An example of loading `stm3110` specific Kconfig files in this structure:

```
soc
└── arm
    └── st_stm32
            ├── Kconfig.soc
            └── stm3210
                └── Kconfig.series
```

can be done with the following content in `st_stm32/Kconfig.soc`:

```
rsource "*/Kconfig.series"
```

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

Using west:

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

Using CMake and ninja:

```
cmake -B build -GNinja -DBOARD=<board name> -DSOC_ROOT=<path to soc> -DBOARD_ROOT=
↪<path to boards> .
ninja -C build
```

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

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

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

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

6.14.3 Devicetree Definitions

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

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

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

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

Using west:

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

Using CMake and ninja:

```
cmake -B build -GNinja -DBOARD=<board name> -DDTS_ROOT=<path to dts root> .
ninja -C build
```

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

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

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

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

Using west:

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

Using CMake and ninja:

```
cmake -B build -GNinja -DBOARD=<board name> -DDTS_EXTRA_CPPFLAGS=-DTEST_ENABLE_
↪FEATURE .
ninja -C build
```

6.15 Debug with Eclipse

6.15.1 Overview

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

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

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

6.15.2 Set Up the Eclipse Development Environment

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

6.15.3 Generate and Import an Eclipse Project

1. Set up a GNU Arm Embedded toolchain as described in *3rd Party Toolchains*.
2. Navigate to a folder outside of the Zephyr tree to build your application.

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

Note: If the build directory is a subdirectory of the source directory, as is usually done in Zephyr, CMake will warn:

"The build directory is a subdirectory of the source directory.

This is not supported well by Eclipse. It is strongly recommended to use a build directory which is a sibling of the source directory."

---

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

   Using west:

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

   Using CMake and ninja:

```
cmake -B build -GNinja -DBOARD=frdm_k64f -G"Eclipse CDT4 - Ninja" %ZEPHYR_BASE%\
↪samples\synchronization
ninja -C build
```

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

6.15.4 Create a Debugger Configuration

1. Open the menu `Run->Debug Configurations...`.
2. Select `GDB PyOCD Debugging`, click the `New` button, and configure the following options:
   - In the Main tab:
     - Project: my_zephyr_app@build
     - C/C++ Application: `zephyr/zephyr.elf`
   - In the Debugger tab:

  - pyOCD Setup
    - Executable path: $*pyocd_path*\$*pyocd_executable*
    - Uncheck “Allocate console for semihosting”
  - Board Setup
    - Bus speed: 8000000 Hz
    - Uncheck “Enable semihosting”
  - GDB Client Setup
    - Executable path example (use your `GNUARMEMB_TOOLCHAIN_PATH`): `C:\gcc-arm-none-eabi-6_2017-q2-update\bin\arm-none-eabi-gdb.exe`
- In the SVD Path tab:
  - File path: `<workspace top>\modules\hal\nxp\mcux\devices\MK64F12\MK64F12.xml`

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

---

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

6.15.5 RTOS Awareness

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

Chapter 7

API Reference

7.1 API Status / Guidelines

7.1.1 API Overview

The table lists Zephyr's APIs and information about them, including their current *stability level*.

API	Status	Version Introduced	Version Modified
ADC	Stable	1.0	2.6
Audio Codec	Experimental	1.13	1.13
Audio DMIC	Experimental	1.13	1.13
Bluetooth	Stable	1.0	2.4
Clock Control	Stable	1.0	2.6
CoAP	Unstable	1.10	2.4
Controller Area Network (CAN)	Unstable	1.14	2.6
Counter	Unstable	1.14	2.6
Crypto	Stable	1.7	2.2
DAC	Experimental	2.3	2.3
DMA	Stable	1.5	2.6
Device Driver Model	Stable	1.0	2.4
Devicetree API	Stable	2.2	2.6
Disk Access	Stable	1.6	2.0
Display Interface	Unstable	1.14	2.2
EC Host Command	Experimental	2.4	2.4
Error Detection And Correction (EDAC) API	Experimental	2.5	2.5
EEPROM	Stable	2.1	2.1
Entropy	Stable	1.10	1.12
File Systems	Stable	1.5	2.4
Flash	Stable	1.2	2.6
Flash Circular Buffer (FCB)	Stable	1.11	2.1
Flash map	Stable	1.11	2.6
GNA	Experimental	1.14	1.14
GPIO	Stable	1.0	2.6
Hardware Information	Stable	1.14	2.3
I2C EEPROM Slave	Stable	1.13	1.13
I2C	Stable	1.0	2.6
I2C Slave API	Experimental	1.12	1.12
I2S	Stable	1.9	2.6
IPM	Stable	1.0	2.4

continues on next page

Table 1 – continued from previous page

API	Status	Version Introduced	Version Modified
KSCAN	Stable	2.1	2.6
Kernel Services	Stable	1.0	2.6
LED	Stable	1.12	2.6
Lightweight M2M (LWM2M)	Unstable	1.9	2.5
Logging	Stable	1.13	1.14
MQTT	Unstable	1.14	2.4
Miscellaneous APIs	Stable	1.0	2.2
Networking	Stable	1.0	2.4
Non-Volatile Storage (NVS)	Stable	1.12	1.14
PECI	Stable	2.1	2.6
PS/2	Stable	2.1	2.6
PWM	Stable	1.0	2.6
Pinmux	Stable	1.0	1.11
Power Management	Experimental	1.2	2.2
Random Number Generation	Stable	1.0	2.1
Regulators	Experimental	2.4	2.4
SPI	Stable	1.0	2.6
Sensors	Stable	1.2	2.6
Settings	Stable	1.12	2.1
Shell	Stable	1.14	2.4
Stream Flash	Experimental	2.3	2.3
Task Watchdog	Experimental	2.5	2.5
UART	Stable	1.0	2.6
UART async	Unstable	1.14	2.2
USB device support	Stable	1.5	2.4
User Mode	Stable	1.11	1.11
Utilities	Experimental	2.4	2.4
Video	Stable	2.1	2.6
Watchdog	Stable	1.0	2.0

7.1.2 API Lifecycle

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

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

Experimental

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

The following requirements apply to all new APIs:

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

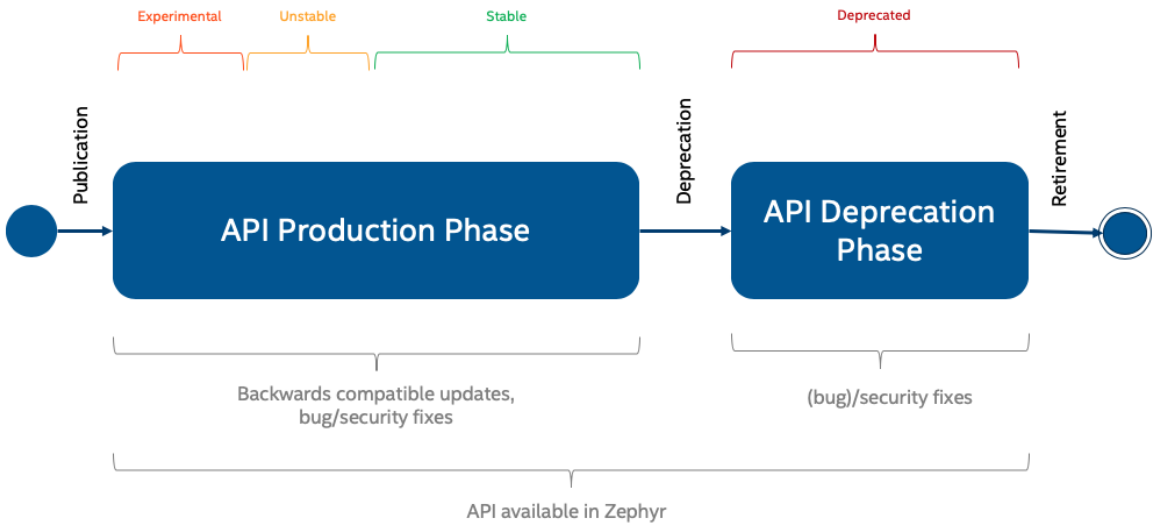


Fig. 1: API Life Cycle

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

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

Unstable

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

Note: Changes will not be announced.

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

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

Stable

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

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

- Test cases for the new API with 100% coverage
- Complete documentation in code. All public interfaces shall be documented and available in online documentation.
- The API has been in-use and was available in at least 2 development releases

- Stable APIs can get backward compatible updates, bug fixes and security fixes at any time.

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

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

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

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

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

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

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

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

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

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

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

- A title that matches the RFC issue
- A link to the RFC issue
- The actual changes to the API
  - Changes to the API header file
  - Changes to the API implementation(s)

  - Changes to the relevant API documentation
  - Changes to Device Tree source and bindings
- The changes required to adapt in-tree users of the API to the change. Depending on the scope of this task this might require additional help from the corresponding maintainers
- An entry in the "API Changes" section of the release notes for the next upcoming release
- The labels `API`, `Stable API Change` and `Release Notes`, as well as any others that are applicable

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

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

Note: Incompatible changes will be announced in the "API Changes" section of the release notes.

Deprecated

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

The following are the requirements for deprecating an existing API:

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

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

- API documentation will inform users that the API is deprecated.
- Attempts to use a deprecated API at build time will log a warning to the console.

Retired

In this phase, the API is removed.

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

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

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

7.1.3 API Design Guidelines

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

Using Callbacks

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

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

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

Examples

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

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

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

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

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

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

Conditional Data and APIs

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

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

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

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

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

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

Return Codes

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

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

7.1.4 API Terminology

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

reschedule if executing the function reaches a reschedule point

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

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

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

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

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

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

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

reschedule

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

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

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

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

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

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

sleep

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

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

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

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

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

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

no-wait

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

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

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

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

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

isr-ok

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

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

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

pre-kernel-ok

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

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

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

async

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

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

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

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

supervisor

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

7.2 Audio

7.2.1 Audio Codec

Overview

The Audio Codec API provides access to digital audio codecs.

Configuration Options

Related configuration options:

- `CONFIG_AUDIO_CODEC`

API Reference

group `audio_codec_interface`

Abstraction for audio codecs.

Enums

enum `audio_pcm_rate_t`

PCM audio sample rates

Values:

enumerator `AUDIO_PCM_RATE_8K` = 8000

enumerator `AUDIO_PCM_RATE_16K` = 16000

enumerator `AUDIO_PCM_RATE_24K` = 24000

enumerator `AUDIO_PCM_RATE_32K` = 32000

enumerator `AUDIO_PCM_RATE_44P1K` = 44100

enumerator `AUDIO_PCM_RATE_48K` = 48000

enumerator `AUDIO_PCM_RATE_96K` = 96000

enumerator `AUDIO_PCM_RATE_192K` = 192000

enum `audio_pcm_width_t`

PCM audio sample bit widths

Values:

enumerator `AUDIO_PCM_WIDTH_16_BITS` = 16

enumerator AUDIO_PCM_WIDTH_20_BITS = 20

enumerator AUDIO_PCM_WIDTH_24_BITS = 24

enumerator AUDIO_PCM_WIDTH_32_BITS = 32

enum audio_dai_type_t

Digital Audio Interface (DAI) type

Values:

enumerator AUDIO_DAI_TYPE_I2S

enumerator AUDIO_DAI_TYPE_INVALID

enum audio_property_t

Codec properties that can be set by *audio_codec_set_property()*

Values:

enumerator AUDIO_PROPERTY_OUTPUT_VOLUME

enumerator AUDIO_PROPERTY_OUTPUT_MUTE

enum audio_channel_t

Audio channel identifiers to use in *audio_codec_set_property()*

Values:

enumerator AUDIO_CHANNEL_FRONT_LEFT

enumerator AUDIO_CHANNEL_FRONT_RIGHT

enumerator AUDIO_CHANNEL_LFE

enumerator AUDIO_CHANNEL_FRONT_CENTER

enumerator AUDIO_CHANNEL_REAR_LEFT

enumerator AUDIO_CHANNEL_REAR_RIGHT

enumerator AUDIO_CHANNEL_REAR_CENTER

enumerator AUDIO_CHANNEL_SIDE_LEFT

enumerator AUDIO_CHANNEL_SIDE_RIGHT

enumerator AUDIO_CHANNEL_ALL

Functions

static inline int `audio_codec_configure`(const struct *device* *dev, struct *audio_codec_cfg* *cfg)

Configure the audio codec.

Configure the audio codec device according to the configuration parameters provided as input

Parameters

- `dev` – Pointer to the device structure for codec driver instance.
- `cfg` – Pointer to the structure containing the codec configuration.

Returns 0 on success, negative error code on failure

static inline void `audio_codec_start_output`(const struct *device* *dev)

Set codec to start output audio playback.

Setup the audio codec device to start the audio playback

Parameters

- `dev` – Pointer to the device structure for codec driver instance.

Returns none

static inline void `audio_codec_stop_output`(const struct *device* *dev)

Set codec to stop output audio playback.

Setup the audio codec device to stop the audio playback

Parameters

- `dev` – Pointer to the device structure for codec driver instance.

Returns none

static inline int `audio_codec_set_property`(const struct *device* *dev, *audio_property_t* property, *audio_channel_t* channel, *audio_property_value_t* val)

Set a codec property defined by audio_property_t.

Set a property such as volume level, clock configuration etc.

Parameters

- `dev` – Pointer to the device structure for codec driver instance.
- `property` – The codec property to set
- `channel` – The audio channel for which the property has to be set
- `val` – pointer to a property value of type audio_codec_property_value_t

Returns 0 on success, negative error code on failure

static inline int `audio_codec_apply_properties`(const struct *device* *dev)

Atomically apply any cached properties.

Following one or more invocations of audio_codec_set_property, that may have been cached by the driver, audio_codec_apply_properties can be invoked to apply all the properties as atomic as possible

Parameters

- `dev` – Pointer to the device structure for codec driver instance.

Returns 0 on success, negative error code on failure

union `audio_dai_cfg_t`

#include <codec.h> Digital Audio Interface Configuration Configuration is dependent on DAI type

Public Members

struct *i2s_config* `i2s`

struct `audio_codec_cfg`

#include <codec.h> Codec configuration parameters

union `audio_property_value_t`

#include <codec.h> Codec property values

Public Members

int `vol`

bool `mute`

7.2.2 Audio DMIC

Overview

The audio DMIC interface provides access to digital microphones.

Configuration Options

Related configuration options:

- `CONFIG_AUDIO_DMIC`

API Reference

group `audio_dmic_interface`

Abstraction for digital microphones.

Enums

enum `dmic_state`

DMIC driver states

Values:

enumerator `DMIC_STATE_UNINIT`

enumerator DMIC_STATE_INITIALIZED

enumerator DMIC_STATE_CONFIGURED

enumerator DMIC_STATE_ACTIVE

enumerator DMIC_STATE_PAUSED

enum dmic_trigger

DMIC driver trigger commands

Values:

enumerator DMIC_TRIGGER_STOP

enumerator DMIC_TRIGGER_START

enumerator DMIC_TRIGGER_PAUSE

enumerator DMIC_TRIGGER_RELEASE

enumerator DMIC_TRIGGER_RESET

enum pdm_lr

PDM Channels LEFT / RIGHT

Values:

enumerator PDM_CHAN_LEFT

enumerator PDM_CHAN_RIGHT

Functions

static inline uint32_t dmic_build_channel_map(uint8_t channel, uint8_t pdm, enum *pdm_lr* lr)

Build the channel map to populate struct *pdm_chan_cfg*

Returns the map of PDM controller and LEFT/RIGHT channel shifted to the bit position corresponding to the input logical channel value

Parameters

- channel – The logical channel number
- pdm – The PDM hardware controller number
- lr – LEFT/RIGHT channel within the chosen PDM hardware controller

Returns Bit-map containing the PDM and L/R channel information

static inline void dmic_parse_channel_map(uint32_t channel_map_lo, uint32_t channel_map_hi, uint8_t channel, uint8_t *pdm, enum *pdm_lr* *lr)

Helper function to parse the channel map in *pdm_chan_cfg*

Returns the PDM controller and LEFT/RIGHT channel corresponding to the channel map and the logical channel provided as input

Parameters

- `channel_map_lo` – Lower order/significant bits of the channel map
- `channel_map_hi` – Higher order/significant bits of the channel map
- `channel` – The logical channel number
- `pdm` – Pointer to the PDM hardware controller number
- `lr` – Pointer to the LEFT/RIGHT channel within the PDM controller

Returns none

static inline uint32_t dmic_build_clk_skew_map(uint8_t pdm, uint8_t skew)

Build a bit map of clock skew values for each PDM channel

Returns the bit-map of clock skew value shifted to the bit position corresponding to the input PDM controller value

Parameters

- `pdm` – The PDM hardware controller number
- `skew` – The skew to apply for the clock output from the PDM controller

Returns Bit-map containing the clock skew information

static inline int dmic_configure(const struct *device* *dev, struct *dmic_cfg* *cfg)

Configure the DMIC driver and controller(s)

Configures the DMIC driver device according to the number of channels, channel mapping, PDM I/O configuration, PCM stream configuration, etc.

Parameters

- `dev` – Pointer to the device structure for DMIC driver instance
- `cfg` – Pointer to the structure containing the DMIC configuration

Returns 0 on success, a negative error code on failure

static inline int dmic_trigger(const struct *device* *dev, enum *dmic_trigger* cmd)

Send a command to the DMIC driver

Sends a command to the driver to perform a specific action

Parameters

- `dev` – Pointer to the device structure for DMIC driver instance
- `cmd` – The command to be sent to the driver instance

Returns 0 on success, a negative error code on failure

static inline int dmic_read(const struct *device* *dev, uint8_t stream, void **buffer, size_t *size, int32_t timeout)

Read received decimated PCM data stream

Optionally waits for audio to be received and provides the received audio buffer from the requested stream

Parameters

- `dev` – Pointer to the device structure for DMIC driver instance

- `stream` – Stream identifier
- `buffer` – Pointer to the received buffer address
- `size` – Pointer to the received buffer size
- `timeout` – Timeout in milliseconds to wait in case audio is not yet received, or SYS_FOREVER_MS

Returns 0 on success, a negative error code on failure

struct `pdm_io_cfg`

#include <*dmic.h*> PDM Input/Output signal configuration

struct `pcm_stream_cfg`

#include <*dmic.h*> Configuration of the PCM streams to be output by the PDM hardware

struct `pdm_chan_cfg`

#include <*dmic.h*> Mapping/ordering of the PDM channels to logical PCM output channel

struct `dmic_cfg`

#include <*dmic.h*> Input configuration structure for the DMIC configuration API

7.2.3 I2S

Overview

The I2S (Inter-IC Sound) API provides support for the standard I2S interface as well as common non-standard extensions such as PCM Short/Long Frame Sync and Left/Right Justified Data Formats.

Configuration Options

Related configuration options:

- `CONFIG_I2S`

API Reference

group `i2s_interface`

I2S (Inter-IC Sound) Interface.

The I2S API provides support for the standard I2S interface standard as well as common non-standard extensions such as PCM Short/Long Frame Sync, Left/Right Justified Data Format.

Defines

`I2S_FMT_DATA_FORMAT_SHIFT`

Data Format bit field position.

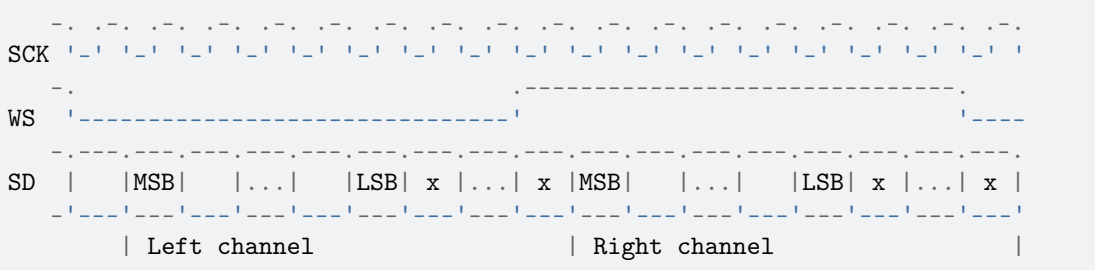
`I2S_FMT_DATA_FORMAT_MASK`

Data Format bit field mask.

I2S_FMT_DATA_FORMAT_I2S

Standard I2S Data Format.

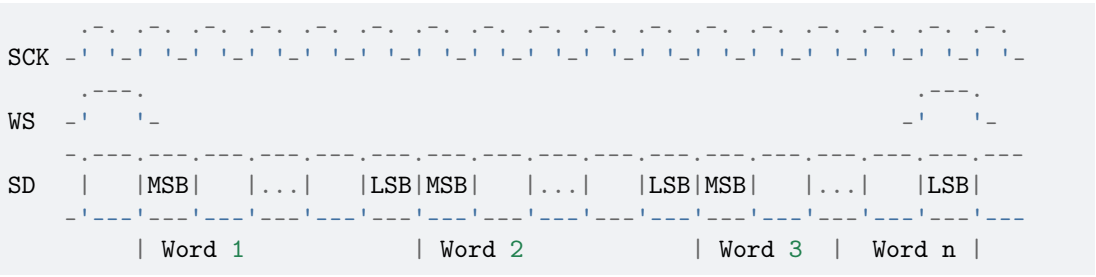
Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the rising edge of the clock signal (SCK). The MSB is always sent one clock period after the WS changes. Left channel data are sent first indicated by WS = 0, followed by right channel data indicated by WS = 1.



I2S_FMT_DATA_FORMAT_PCM_SHORT

PCM Short Frame Sync Data Format.

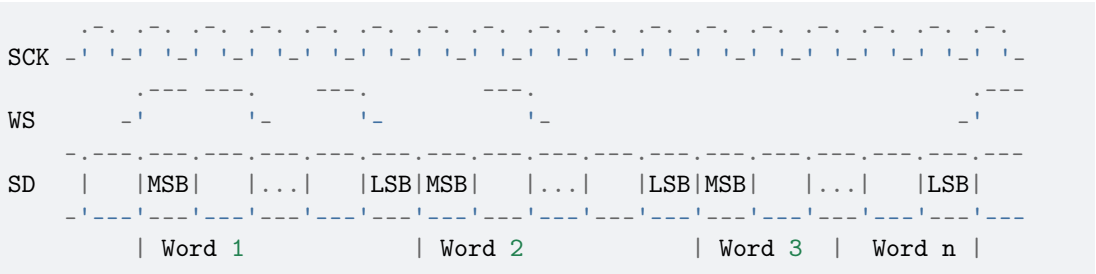
Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the falling edge of the clock signal (SCK). The falling edge of the frame sync signal (WS) indicates the start of the PCM word. The frame sync is one clock cycle long. An arbitrary number of data words can be sent in one frame.



I2S_FMT_DATA_FORMAT_PCM_LONG

PCM Long Frame Sync Data Format.

Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the falling edge of the clock signal (SCK). The rising edge of the frame sync signal (WS) indicates the start of the PCM word. The frame sync has an arbitrary length, however it has to fall before the start of the next frame. An arbitrary number of data words can be sent in one frame.

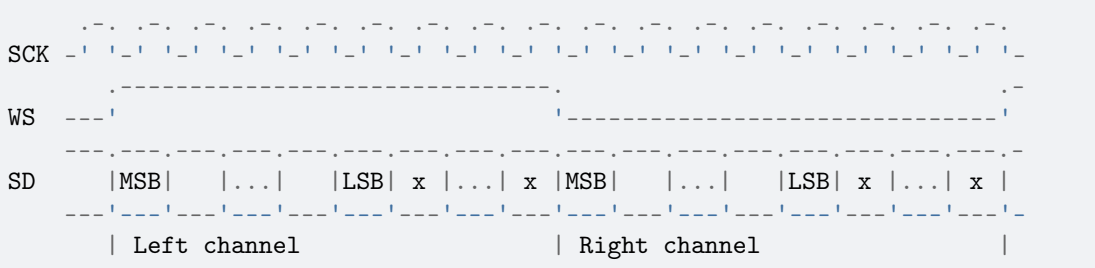


I2S_FMT_DATA_FORMAT_LEFT_JUSTIFIED

Left Justified Data Format.

Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the rising edge of the clock signal (SCK). The

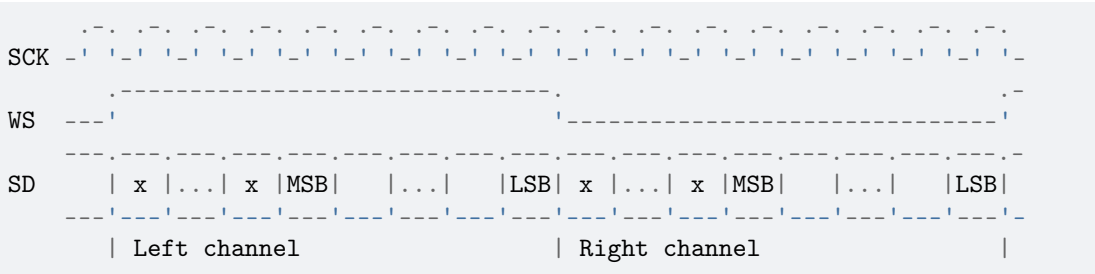
bits within the data word are left justified such that the MSB is always sent in the clock period following the WS transition. Left channel data are sent first indicated by WS = 1, followed by right channel data indicated by WS = 0.



I2S_FMT_DATA_FORMAT_RIGHT_JUSTIFIED

Right Justified Data Format.

Serial data is transmitted in two's complement with the MSB first. Both Word Select (WS) and Serial Data (SD) signals are sampled on the rising edge of the clock signal (SCK). The bits within the data word are right justified such that the LSB is always sent in the clock period preceding the WS transition. Left channel data are sent first indicated by WS = 1, followed by right channel data indicated by WS = 0.



I2S_FMT_DATA_ORDER_MSB

Send MSB first

I2S_FMT_DATA_ORDER_LSB

Send LSB first

I2S_FMT_DATA_ORDER_INV

Invert bit ordering, send LSB first

I2S_FMT_CLK_FORMAT_SHIFT

Data Format bit field position.

I2S_FMT_CLK_FORMAT_MASK

Data Format bit field mask.

I2S_FMT_BIT_CLK_INV

Invert bit clock

I2S_FMT_FRAME_CLK_INV

Invert frame clock

I2S_FMT_CLK_NF_NB

NF represents “Normal Frame” whereas IF represents “Inverted Frame” NB represents “Normal Bit Clk” whereas IB represents “Inverted Bit clk”

I2S_FMT_CLK_NF_IB

I2S_FMT_CLK_IF_NB

I2S_FMT_CLK_IF_IB

I2S_OPT_BIT_CLK_CONT

Run bit clock continuously

I2S_OPT_BIT_CLK_GATED

Run bit clock when sending data only

I2S_OPT_BIT_CLK_MASTER

I2S driver is bit clock master

I2S_OPT_BIT_CLK_SLAVE

I2S driver is bit clock slave

I2S_OPT_FRAME_CLK_MASTER

I2S driver is frame clock master

I2S_OPT_FRAME_CLK_SLAVE

I2S driver is frame clock slave

I2S_OPT_LOOPBACK

Loop back mode.

In loop back mode RX input will be connected internally to TX output. This is used primarily for testing.

I2S_OPT_PINGPONG

Ping pong mode.

In ping pong mode TX output will keep alternating between a ping buffer and a pong buffer. This is normally used in audio streams when one buffer is being populated while the other is being played (DMAed) and vice versa. So, in this mode, 2 sets of buffers fixed in size are used. Static Arrays are used to achieve this and hence they are never freed.

Typedefs

typedef uint8_t i2s_fmt_t

typedef uint8_t i2s_opt_t

Enums

enum i2s_dir

I2C Direction.

Values:

enumerator I2S_DIR_RX

Receive data

enumerator I2S_DIR_TX

Transmit data

enumerator I2S_DIR_BOTH

Both receive and transmit data

enum i2s_state

Interface state

Values:

enumerator I2S_STATE_NOT_READY

The interface is not ready.

```
The interface was initialized but is not yet ready to receive /
transmit data. Call i2s_configure() to configure interface and change
its state to READY.
```

enumerator I2S_STATE_READY

The interface is ready to receive / transmit data.

enumerator I2S_STATE_RUNNING

The interface is receiving / transmitting data.

enumerator I2S_STATE_STOPPING

The interface is draining its transmit queue.

enumerator I2S_STATE_ERROR

TX buffer underrun or RX buffer overrun has occurred.

enum i2s_trigger_cmd

Trigger command

Values:

enumerator I2S_TRIGGER_START

Start the transmission / reception of data.

```
If I2S_DIR_TX is set some data has to be queued for transmission by
the i2s_write() function. This trigger can be used in READY state
only and changes the interface state to RUNNING.
```

enumerator I2S_TRIGGER_STOP

Stop the transmission / reception of data.

```
Stop the transmission / reception of data at the end of the current
memory block. This trigger can be used in RUNNING state only and at
first changes the interface state to STOPPING. When the current TX /
RX block is transmitted / received the state is changed to READY.
Subsequent START trigger will resume transmission / reception where
it stopped.
```

enumerator I2S_TRIGGER_DRAIN

Empty the transmit queue.

```
Send all data in the transmit queue and stop the transmission.
If the trigger is applied to the RX queue it has the same effect as
I2S_TRIGGER_STOP. This trigger can be used in RUNNING state only and
at first changes the interface state to STOPPING. When all TX blocks
are transmitted the state is changed to READY.
```

enumerator I2S_TRIGGER_DROP

Discard the transmit / receive queue.

```
Stop the transmission / reception immediately and discard the
contents of the respective queue. This trigger can be used in any
state other than NOT_READY and changes the interface state to READY.
```

enumerator I2S_TRIGGER_PREPARE

Prepare the queues after underrun/overrun error has occurred.

```
This trigger can be used in ERROR state only and changes the
interface state to READY.
```

Functions

int i2s_configure(const struct *device* *dev, enum *i2s_dir* dir, const struct *i2s_config* *cfg)

Configure operation of a host I2S controller.

The dir parameter specifies if Transmit (TX) or Receive (RX) direction will be configured by data provided via cfg parameter.

The function can be called in NOT_READY or READY state only. If executed successfully the function will change the interface state to READY.

If the function is called with the parameter cfg->frame_clk_freq set to 0 the interface state will be changed to NOT_READY.

Parameters

- dev – Pointer to the device structure for the driver instance.
- dir – Stream direction: RX, TX, or both, as defined by I2S_DIR_*. The I2S_DIR_BOTH value may not be supported by some drivers. For those, the RX and TX streams need to be configured separately.
- cfg – Pointer to the structure containing configuration parameters.

Return values

- `0` – If successful.
- `-EINVAL` – Invalid argument.
- `-ENOSYS` – I2S_DIR_BOTH value is not supported.

static inline const struct *i2s_config* *`i2s_config_get`(const struct *device* *dev, enum *i2s_dir* dir)

Fetch configuration information of a host I2S controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `dir` – Stream direction: RX or TX as defined by I2S_DIR_*

Return values `Pointer` – to the structure containing configuration parameters, or NULL if un-configured

static inline int `i2s_read`(const struct *device* *dev, void **mem_block, size_t *size)

Read data from the RX queue.

Data received by the I2S interface is stored in the RX queue consisting of memory blocks preallocated by this function from rx_mem_slab (as defined by i2s_configure). Ownership of the RX memory block is passed on to the user application which has to release it.

The data is read in chunks equal to the size of the memory block. If the interface is in READY state the number of bytes read can be smaller.

If there is no data in the RX queue the function will block waiting for the next RX memory block to fill in. This operation can timeout as defined by i2s_configure. If the timeout value is set to K_NO_WAIT the function is non-blocking.

Reading from the RX queue is possible in any state other than NOT_READY. If the interface is in the ERROR state it is still possible to read all the valid data stored in RX queue. Afterwards the function will return -EIO error.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mem_block` – Pointer to the RX memory block containing received data.
- `size` – Pointer to the variable storing the number of bytes read.

Return values

- `0` – If successful.
- `-EIO` – The interface is in NOT_READY or ERROR state and there are no more data blocks in the RX queue.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

int `i2s_buf_read`(const struct *device* *dev, void *buf, size_t *size)

Read data from the RX queue into a provided buffer.

Data received by the I2S interface is stored in the RX queue consisting of memory blocks preallocated by this function from rx_mem_slab (as defined by i2s_configure). Calling this function removes one block from the queue which is copied into the provided buffer and then freed.

The provided buffer must be large enough to contain a full memory block of data, which is parameterized for the channel via *i2s_configure()*.

This function is otherwise equivalent to *i2s_read()*.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

- `buf` – Destination buffer for read data, which must be at least the as large as the configured memory block size for the RX channel.
- `size` – Pointer to the variable storing the number of bytes read.

Return values

- `0` – If successful.
- `-EIO` – The interface is in NOT_READY or ERROR state and there are no more data blocks in the RX queue.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

static inline int `i2s_write`(const struct *device* *dev, void *mem_block, size_t size)

Write data to the TX queue.

Data to be sent by the I2S interface is stored first in the TX queue. TX queue consists of memory blocks preallocated by the user from tx_mem_slab (as defined by i2s_configure). This function takes ownership of the memory block and will release it when all data are transmitted.

If there are no free slots in the TX queue the function will block waiting for the next TX memory block to be send and removed from the queue. This operation can timeout as defined by i2s_configure. If the timeout value is set to K_NO_WAIT the function is non-blocking.

Writing to the TX queue is only possible if the interface is in READY or RUNNING state.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mem_block` – Pointer to the TX memory block containing data to be sent.
- `size` – Number of bytes to write. This value has to be equal or smaller than the size of the memory block.

Return values

- `0` – If successful.
- `-EIO` – The interface is not in READY or RUNNING state.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

int `i2s_buf_write`(const struct *device* *dev, void *buf, size_t size)

Write data to the TX queue from a provided buffer.

This function acquires a memory block from the I2S channel TX queue and copies the provided data buffer into it. It is otherwise equivalent to *i2s_write()*.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `buf` – Pointer to a buffer containing the data to transmit.
- `size` – Number of bytes to write. This value has to be equal or smaller than the size of the channel's TX memory block configuration.

Return values

- `0` – If successful.
- `-EIO` – The interface is not in READY or RUNNING state.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

- `-ENOMEM` – No memory in TX slab queue.
- `-EINVAL` – Size parameter larger than TX queue memory block.

int `i2s_trigger`(const struct *device* *dev, enum *i2s_dir* dir, enum *i2s_trigger_cmd* cmd)

Send a trigger command.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `dir` – Stream direction: RX, TX, or both, as defined by I2S_DIR_*. The I2S_DIR_BOTH value may not be supported by some drivers. For those, triggering need to be done separately for the RX and TX streams.
- `cmd` – Trigger command.

Return values

- `0` – If successful.
- `-EINVAL` – Invalid argument.
- `-EIO` – The trigger cannot be executed in the current state or a DMA channel cannot be allocated.
- `-ENOMEM` – RX/TX memory block not available.
- `-ENOSYS` – I2S_DIR_BOTH value is not supported.

struct `i2s_config`

#include <i2s.h> Interface configuration options.

Memory slab pointed to by the mem_slab field has to be defined and initialized by the user. For I2S driver to function correctly number of memory blocks in a slab has to be at least 2 per queue. Size of the memory block should be multiple of frame_size where frame_size = (channels * word_size_bytes). As an example 16 bit word will occupy 2 bytes, 24 or 32 bit word will occupy 4 bytes.

Please check Zephyr Kernel Primer for more information on memory slabs.

Remark

When I2S data format is selected parameter channels is ignored, number of words in a frame is always 2.

Param word_size Number of bits representing one data word.

Param channels Number of words per frame.

Param format Data stream format as defined by I2S_FMT_* constants.

Param options Configuration options as defined by I2S_OPT_* constants.

Param frame_clk_freq Frame clock (WS) frequency, this is sampling rate.

Param mem_slab memory slab to store RX/TX data.

Param block_size Size of one RX/TX memory block (buffer) in bytes.

Param timeout Read/Write timeout. Number of milliseconds to wait in case TX queue is full or RX queue is empty, or 0, or SYS_FOREVER_MS.

7.3 Asynchronous Notification APIs

Zephyr APIs often include *async* functions where an operation is initiated and the application needs to be informed when it completes, and whether it succeeded. Using *k_poll()* is often a good method, but some application architectures may be more suited to a callback notification, and operations like enabling clocks and power rails may need to be invoked before kernel functions are available so a busy-wait for completion may be needed.

This API is intended to be embedded within specific subsystems such as *On-Off Manager* and other APIs that support async transactions. The subsystem wrappers are responsible for extracting operation-specific data from requests that include a notification element, and for invoking callbacks with the parameters required by the API.

A limitation is that this API is not suitable for *System Calls* because:

- *sys_notify* is not a kernel object;
- copying the notification content from userspace will break use of *CONTAINER_OF* in the implementing function;
- neither the spin-wait nor callback notification methods can be accepted from userspace callers.

Where a notification is required for an asynchronous operation invoked from a user mode thread the subsystem or driver should provide a syscall API that uses *k_poll_signal* for notification.

7.3.1 API Reference

group sys_notify_apis

Typedefs

typedef void (*sys_notify_generic_callback)()

Generic signature used to notify of result completion by callback.

Functions with this role may be invoked from any context including pre-kernel, ISR, or cooperative or pre-emptible threads. Compatible functions must be isr-ok and not sleep.

Parameters that should generally be passed to such functions include:

- a pointer to a specific client request structure, i.e. the one that contains the *sys_notify* structure.
- the result of the operation, either as passed to *sys_notify_finalize()* or extracted afterwards using *sys_notify_fetch_result()*. Expected values are service-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed.

Functions

static inline uint32_t sys_notify_get_method(const struct *sys_notify* *notify)

int sys_notify_validate(struct *sys_notify* *notify)

Validate and initialize the notify structure.

This should be invoked at the start of any service-specific configuration validation. It ensures that the basic asynchronous notification configuration is consistent, and clears the result.

Note that this function does not validate extension bits (zeroed by async notify API init functions like *sys_notify_init_callback()*). It may fail to recognize that an uninitialized structure has been passed because only method bits of flags are tested against method settings. To reduce the chance of accepting an uninititalized operation service validation of structures that contain an *sys_notify* instance should confirm that the extension bits are set or cleared as expected.

Return values

- `0` – on successful validation and reinitialization
- `-EINVAL` – if the configuration is not valid.

sys_notify_generic_callback `sys_notify_finalize`(struct *sys_notify* *notify, int res)

Record and signal the operation completion.

Parameters

- `notify` – pointer to the notification state structure.
- `res` – the result of the operation. Expected values are service-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed.

Returns If the notification is to be done by callback this returns the generic version of the function to be invoked. The caller must immediately invoke that function with whatever arguments are expected by the callback. If notification is by spin-wait or signal, the notification has been completed by the point this function returns, and a null pointer is returned.

static inline int `sys_notify_fetch_result`(const struct *sys_notify* *notify, int *result)

Check for and read the result of an asynchronous operation.

Parameters

- `notify` – pointer to the object used to specify asynchronous function behavior and store completion information.
- `result` – pointer to storage for the result of the operation. The result is stored only if the operation has completed.

Return values

- `0` – if the operation has completed.
- `-EAGAIN` – if the operation has not completed.

static inline void `sys_notify_init_spinwait`(struct *sys_notify* *notify)

Initialize a notify object for spin-wait notification.

Clients that use this initialization receive no asynchronous notification, and instead must periodically check for completion using *sys_notify_fetch_result()*.

On completion of the operation the client object must be reinitialized before it can be re-used.

Parameters

- `notify` – pointer to the notification configuration object.

static inline void `sys_notify_init_signal`(struct *sys_notify* *notify, struct *k_poll_signal* *sigp)

Initialize a notify object for (k_poll) signal notification.

Clients that use this initialization will be notified of the completion of operations through the provided signal.

On completion of the operation the client object must be reinitialized before it can be re-used.

Note: This capability is available only when `CONFIG_POLL` is selected.

Parameters

- `notify` – pointer to the notification configuration object.
- `sigp` – pointer to the signal to use for notification. The value must not be null. The signal must be reset before the client object is passed to the on-off service API.

static inline void `sys_notify_init_callback`(struct *sys_notify* *notify, *sys_notify_generic_callback* handler)

Initialize a notify object for callback notification.

Clients that use this initialization will be notified of the completion of operations through the provided callback. Note that callbacks may be invoked from various contexts depending on the specific service; see *sys_notify_generic_callback*.

On completion of the operation the client object must be reinitialized before it can be re-used.

Parameters

- `notify` – pointer to the notification configuration object.
- `handler` – a function pointer to use for notification.

static inline bool `sys_notify_uses_callback`(const struct *sys_notify* *notify)

Detect whether a particular notification uses a callback.

The generic handler does not capture the signature expected by the callback, and the translation to a service-specific callback must be provided by the service. This check allows abstracted services to reject callback notification requests when the service doesn't provide a translation function.

Returns true if and only if a callback is to be used for notification.

struct `sys_notify`

#include <notify.h> State associated with notification for an asynchronous operation.

Objects of this type are allocated by a client, which must use an initialization function (e.g. *sys_notify_init_signal()*) to configure them. Generally the structure is a member of a service-specific client structure, such as *onoff_client*.

Control of the containing object transfers to the service provider when a pointer to the object is passed to a service function that is documented to take control of the object, such as onoff_service_request(). While the service provider controls the object the client must not change any object fields. Control reverts to the client:

- if the call to the service API returns an error;
- when operation completion is posted. This may occur before the call to the service API returns.

Operation completion is technically posted when the flags field is updated so that *sys_notify_fetch_result()* returns success. This will happen before the signal is posted or callback is invoked. Note that although the manager will no longer reference the *sys_notify* object past this point, the containing object may have state that will be referenced within the callback. Where callbacks are used control of the containing object does not revert to the client until the callback has been invoked. (Re-use within the callback is explicitly permitted.)

After control has reverted to the client the notify object must be reinitialized for the next operation.

The content of this structure is not public API to clients: all configuration and inspection should be done with functions like *sys_notify_init_callback()* and *sys_notify_fetch_result()*. However, services that use this structure may access certain fields directly.

union `method`

#include <notify.h>

Public Members

struct *k_poll_signal* *`signal`

sys_notify_generic_callback `callback`

7.4 Bluetooth

7.4.1 Connection Management

The Zephyr Bluetooth stack uses an abstraction called `bt_conn` to represent connections to other devices. The internals of this struct are not exposed to the application, but a limited amount of information (such as the remote address) can be acquired using the *bt_conn_get_info()* API. Connection objects are reference counted, and the application is expected to use the *bt_conn_ref()* API whenever storing a connection pointer for a longer period of time, since this ensures that the object remains valid (even if the connection would get disconnected). Similarly the *bt_conn_unref()* API is to be used when releasing a reference to a connection.

An application may track connections by registering a *bt_conn_cb* struct using the *bt_conn_cb_register()* or c:func:*BT_CONN_CB_DEFINE()* APIs. This struct lets the application define callbacks for connection & disconnection events, as well as other events related to a connection such as a change in the security level or the connection parameters. When acting as a central the application will also get hold of the connection object through the return value of the `bt_conn_create_le()` API.

API Reference

group `bt_conn`

Connection management.

Defines

`BT_LE_CONN_PARAM_INIT`(int_min, int_max, lat, to)

Initialize connection parameters.

Parameters

- `int_min` – Minimum Connection Interval (N * 1.25 ms)
- `int_max` – Maximum Connection Interval (N * 1.25 ms)
- `lat` – Connection Latency
- `to` – Supervision Timeout (N * 10 ms)

BT_LE_CONN_PARAM(int_min, int_max, lat, to)

Helper to declare connection parameters inline

Parameters

- int_min – Minimum Connection Interval (N * 1.25 ms)
- int_max – Maximum Connection Interval (N * 1.25 ms)
- lat – Connection Latency
- to – Supervision Timeout (N * 10 ms)

BT_LE_CONN_PARAM_DEFAULT

Default LE connection parameters: Connection Interval: 30-50 ms Latency: 0 Timeout: 4 s

BT_CONN_LE_PHY_PARAM_INIT(_pref_tx_phy, _pref_rx_phy)

Initialize PHY parameters

Parameters

- _pref_tx_phy – Bitmask of preferred transmit PHYs.
- _pref_rx_phy – Bitmask of preferred receive PHYs.

BT_CONN_LE_PHY_PARAM(_pref_tx_phy, _pref_rx_phy)

Helper to declare PHY parameters inline

Parameters

- _pref_tx_phy – Bitmask of preferred transmit PHYs.
- _pref_rx_phy – Bitmask of preferred receive PHYs.

BT_CONN_LE_PHY_PARAM_1M

Only LE 1M PHY

BT_CONN_LE_PHY_PARAM_2M

Only LE 2M PHY

BT_CONN_LE_PHY_PARAM_CODED

Only LE Coded PHY.

BT_CONN_LE_PHY_PARAM_ALL

All LE PHYs.

BT_CONN_LE_DATA_LEN_PARAM_INIT(_tx_max_len, _tx_max_time)

Initialize transmit data length parameters

Parameters

- _tx_max_len – Maximum Link Layer transmission payload size in bytes.
- _tx_max_time – Maximum Link Layer transmission payload time in us.

BT_CONN_LE_DATA_LEN_PARAM(_tx_max_len, _tx_max_time)

Helper to declare transmit data length parameters inline

Parameters

- _tx_max_len – Maximum Link Layer transmission payload size in bytes.
- _tx_max_time – Maximum Link Layer transmission payload time in us.

BT_LE_DATA_LEN_PARAM_DEFAULT

Default LE data length parameters.

BT_LE_DATA_LEN_PARAM_MAX

Maximum LE data length parameters.

BT_CONN_ROLE_MASTER

Connection role (central or peripheral)

BT_CONN_ROLE_SLAVE

BT_CONN_LE_CREATE_PARAM_INIT(_options, _interval, _window)

Initialize create connection parameters.

Parameters

- _options – Create connection options.
- _interval – Create connection scan interval (N * 0.625 ms).
- _window – Create connection scan window (N * 0.625 ms).

BT_CONN_LE_CREATE_PARAM(_options, _interval, _window)

Helper to declare create connection parameters inline

Parameters

- _options – Create connection options.
- _interval – Create connection scan interval (N * 0.625 ms).
- _window – Create connection scan window (N * 0.625 ms).

BT_CONN_LE_CREATE_CONN

Default LE create connection parameters. Scan continuously by setting scan interval equal to scan window.

BT_CONN_LE_CREATE_CONN_AUTO

Default LE create connection using filter accept list parameters. Scan window: 30 ms. Scan interval: 60 ms.

BT_CONN_CB_DEFINE(_name)

Register a callback structure for connection events.

Parameters

- _name – Name of callback structure.

BT_PASSKEY_INVALID

Special passkey value that can be used to disable a previously set fixed passkey.

BT_BR_CONN_PARAM_INIT(role_switch)

Initialize BR/EDR connection parameters.

Parameters

- role_switch – True if role switch is allowed

BT_BR_CONN_PARAM(role_switch)

Helper to declare BR/EDR connection parameters inline

Parameters

- role_switch – True if role switch is allowed

BT_BR_CONN_PARAM_DEFAULT

Default BR/EDR connection parameters: Role switch allowed

Enums

enum [anonymous]

Connection PHY options

Values:

enumerator BT_CONN_LE_PHY_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_CONN_LE_PHY_OPT_CODED_S2 = *BIT*(0)

LE Coded using S=2 coding preferred when transmitting.

enumerator BT_CONN_LE_PHY_OPT_CODED_S8 = *BIT*(1)

LE Coded using S=8 coding preferred when transmitting.

enum [anonymous]

Connection Type

Values:

enumerator BT_CONN_TYPE_LE = *BIT*(0)

LE Connection Type

enumerator BT_CONN_TYPE_BR = *BIT*(1)

BR/EDR Connection Type

enumerator BT_CONN_TYPE_SCO = *BIT*(2)

SCO Connection Type

enumerator BT_CONN_TYPE_ISO = *BIT*(3)

ISO Connection Type

enumerator BT_CONN_TYPE_ALL = *BT_CONN_TYPE_LE* | *BT_CONN_TYPE_BR* | *BT_CONN_TYPE_SCO* | *BT_CONN_TYPE_ISO*

All Connection Type

enum [anonymous]

Values:

enumerator BT_CONN_ROLE_CENTRAL = 0

enumerator BT_CONN_ROLE_PERIPHERAL = 1

enum bt_conn_le_tx_power_phy

Values:

enumerator BT_CONN_LE_TX_POWER_PHY_NONE

Convenience macro for when no PHY is set.

enumerator BT_CONN_LE_TX_POWER_PHY_1M

LE 1M PHY

enumerator BT_CONN_LE_TX_POWER_PHY_2M

LE 2M PHY

enumerator BT_CONN_LE_TX_POWER_PHY_CODED_S8

LE Coded PHY using S=8 coding.

enumerator BT_CONN_LE_TX_POWER_PHY_CODED_S2

LE Coded PHY using S=2 coding.

enum [anonymous]

Values:

enumerator BT_CONN_LE_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_CONN_LE_OPT_CODED = *BIT*(0)

Enable LE Coded PHY.

```
Enable scanning on the LE Coded PHY.
```

enumerator BT_CONN_LE_OPT_NO_1M = *BIT*(1)

Disable LE 1M PHY.

```
Disable scanning on the LE 1M PHY.

@note Requires @ref BT_CONN_LE_OPT_CODED.
```

enum bt_security_t

Security level.

Values:

enumerator BT_SECURITY_L0

Level 0: Only for BR/EDR special cases, like SDP

enumerator BT_SECURITY_L1

Level 1: No encryption and no authentication.

enumerator BT_SECURITY_L2
Level 2: Encryption and no authentication (no MITM).

enumerator BT_SECURITY_L3
Level 3: Encryption and authentication (MITM).

enumerator BT_SECURITY_L4
Level 4: Authenticated Secure Connections and 128-bit key.

enumerator BT_SECURITY_FORCE_PAIR = *BIT*(7)
Bit to force new pairing procedure, bit-wise OR with requested security level.

enum bt_security_err
Values:

enumerator BT_SECURITY_ERR_SUCCESS
Security procedure successful.

enumerator BT_SECURITY_ERR_AUTH_FAIL
Authentication failed.

enumerator BT_SECURITY_ERR_PIN_OR_KEY_MISSING
PIN or encryption key is missing.

enumerator BT_SECURITY_ERR_OOB_NOT_AVAILABLE
OOB data is not available.

enumerator BT_SECURITY_ERR_AUTH_REQUIREMENT
The requested security level could not be reached.

enumerator BT_SECURITY_ERR_PAIR_NOT_SUPPORTED
Pairing is not supported

enumerator BT_SECURITY_ERR_PAIR_NOT_ALLOWED
Pairing is not allowed.

enumerator BT_SECURITY_ERR_INVALID_PARAM
Invalid parameters.

enumerator BT_SECURITY_ERR_KEY_REJECTED
Distributed Key Rejected

enumerator BT_SECURITY_ERR_UNSPECIFIED
Pairing failed but the exact reason could not be specified.

Functions

struct bt_conn *bt_conn_ref(struct bt_conn *conn)

Increment a connection's reference count.

Increment the reference count of a connection object.

Note: Will return NULL if the reference count is zero.

Parameters

- conn – Connection object.

Returns Connection object with incremented reference count, or NULL if the reference count is zero.

void bt_conn_unref(struct bt_conn *conn)

Decrement a connection's reference count.

Decrement the reference count of a connection object.

Parameters

- conn – Connection object.

void bt_conn_foreach(int type, void (*func)(struct bt_conn *conn, void *data), void *data)

Iterate through all existing connections.

Parameters

- type – Connection Type
- func – Function to call for each connection.
- data – Data to pass to the callback function.

struct bt_conn *bt_conn_lookup_addr_le(uint8_t id, const *bt_addr_le_t* *peer)

Look up an existing connection by address.

Look up an existing connection based on the remote address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Parameters

- id – Local identity (in most cases BT_ID_DEFAULT).
- peer – Remote address.

Returns Connection object or NULL if not found.

const *bt_addr_le_t* *bt_conn_get_dst(const struct bt_conn *conn)

Get destination (peer) address of a connection.

Parameters

- conn – Connection object.

Returns Destination address.

uint8_t bt_conn_index(struct bt_conn *conn)

Get array index of a connection.

This function is used to map bt_conn to index of an array of connections. The array has CONFIG_BT_MAX_CONN elements.

Parameters

- conn – Connection object.

Returns Index of the connection object. The range of the returned value is 0..CONFIG_BT_MAX_CONN-1

int `bt_conn_get_info`(const struct bt_conn *conn, struct *bt_conn_info* *info)

Get connection info.

Parameters

- `conn` – Connection object.
- `info` – Connection info object.

Returns Zero on success or (negative) error code on failure.

int `bt_conn_get_remote_info`(struct bt_conn *conn, struct *bt_conn_remote_info* *remote_info)

Get connection info for the remote device.

Note: In order to retrieve the remote version (version, manufacturer and subversion) `CONFIG_BT_REMOTE_VERSION` must be enabled

---

Note: The remote information is exchanged directly after the connection has been established. The application can be notified about when the remote information is available through the remote_info_available callback.

Parameters

- `conn` – Connection object.
- `remote_info` – Connection remote info object.

Returns Zero on success or (negative) error code on failure.

Returns -EBUSY The remote information is not yet available.

int `bt_conn_le_get_tx_power_level`(struct bt_conn *conn, struct *bt_conn_le_tx_power* *tx_power_level)

Get connection transmit power level.

Parameters

- `conn` – Connection object.
- `tx_power_level` – Transmit power level descriptor.

Returns Zero on success or (negative) error code on failure.

Returns -ENOBUFS HCI command buffer is not available.

int `bt_conn_le_param_update`(struct bt_conn *conn, const struct *bt_le_conn_param* *param)

Update the connection parameters.

If the local device is in the peripheral role then updating the connection parameters will be delayed. This delay can be configured by through the `CONFIG_BT_CONN_PARAM_UPDATE_TIMEOUT` option.

Parameters

- `conn` – Connection object.
- `param` – Updated connection parameters.

Returns Zero on success or (negative) error code on failure.

int bt_conn_le_data_len_update(struct bt_conn *conn, const struct *bt_conn_le_data_len_param* *param)

Update the connection transmit data length parameters.

Parameters

- conn – Connection object.
- param – Updated data length parameters.

Returns Zero on success or (negative) error code on failure.

int bt_conn_le_phy_update(struct bt_conn *conn, const struct *bt_conn_le_phy_param* *param)

Update the connection PHY parameters.

Update the preferred transmit and receive PHYs of the connection. Use *BT_GAP_LE_PHY_NONE* to indicate no preference.

Parameters

- conn – Connection object.
- param – Updated connection parameters.

Returns Zero on success or (negative) error code on failure.

int bt_conn_disconnect(struct bt_conn *conn, uint8_t reason)

Disconnect from a remote device or cancel pending connection.

Disconnect an active connection with the specified reason code or cancel pending outgoing connection.

The disconnect reason for a normal disconnect should be: BT_HCI_ERR_REMOTE_USER_TERM_CONN.

The following disconnect reasons are accepted:

- BT_HCI_ERR_AUTH_FAIL
- BT_HCI_ERR_REMOTE_USER_TERM_CONN
- BT_HCI_ERR_REMOTE_LOW_RESOURCES
- BT_HCI_ERR_REMOTE_POWER_OFF
- BT_HCI_ERR_UNSUPP_REMOTE_FEATURE
- BT_HCI_ERR_PAIRING_NOT_SUPPORTED
- BT_HCI_ERR_UNACCEPT_CONN_PARAM

Parameters

- conn – Connection to disconnect.
- reason – Reason code for the disconnection.

Returns Zero on success or (negative) error code on failure.

int bt_conn_le_create(const *bt_addr_le_t* *peer, const struct *bt_conn_le_create_param* *create_param, const struct *bt_le_conn_param* *conn_param, struct bt_conn **conn)

Initiate an LE connection to a remote device.

Allows initiate new LE link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

This uses the General Connection Establishment procedure.

The application must disable explicit scanning before initiating a new LE connection.

Parameters

- `peer` – **[in]** Remote address.
- `create_param` – **[in]** Create connection parameters.
- `conn_param` – **[in]** Initial connection parameters.
- `conn` – **[out]** Valid connection object on success.

Returns Zero on success or (negative) error code on failure.

int `bt_conn_le_create_auto`(const struct *bt_conn_le_create_param* *create_param, const struct *bt_le_conn_param* *conn_param)

Automatically connect to remote devices in the filter accept list..

This uses the Auto Connection Establishment procedure. The procedure will continue until a single connection is established or the procedure is stopped through *bt_conn_create_auto_stop*. To establish connections to all devices in the the filter accept list the procedure should be started again in the connected callback after a new connection has been established.

Parameters

- `create_param` – Create connection parameters
- `conn_param` – Initial connection parameters.

Returns Zero on success or (negative) error code on failure.

Returns -ENOMEM No free connection object available.

int `bt_conn_create_auto_stop`(void)

Stop automatic connect creation.

Returns Zero on success or (negative) error code on failure.

int `bt_le_set_auto_conn`(const *bt_addr_le_t* *addr, const struct *bt_le_conn_param* *param)

Automatically connect to remote device if it's in range.

This function enables/disables automatic connection initiation. Every time the device loses the connection with peer, this connection will be re-established if connectable advertisement from peer is received.

Note: Auto connect is disabled during explicit scanning.

Parameters

- `addr` – Remote Bluetooth address.
- `param` – If non-NULL, auto connect is enabled with the given parameters. If NULL, auto connect is disabled.

Returns Zero on success or error code otherwise.

int `bt_conn_set_security`(struct bt_conn *conn, *bt_security_t* sec)

Set security level for a connection.

This function enable security (encryption) for a connection. If the device has bond information for the peer with sufficiently strong key encryption will be enabled. If the connection is already encrypted with sufficiently strong key this function does nothing.

If the device has no bond information for the peer and is not already paired then the pairing procedure will be initiated. If the device has bond information or is already paired and the keys are too weak then the pairing procedure will be initiated.

This function may return error if required level of security is not possible to achieve due to local or remote device limitation (e.g., input output capabilities), or if the maximum number of paired devices has been reached.

This function may return error if the pairing procedure has already been initiated by the local device or the peer device.

Note: When `CONFIG_BT_SMP_SC_ONLY` is enabled then the security level will always be level 4.

Note: When `CONFIG_BT_SMP_OOB_LEGACY_PAIR_ONLY` is enabled then the security level will always be level 3.

Parameters

- `conn` – Connection object.
- `sec` – Requested security level.

Returns 0 on success or negative error

bt_security_t `bt_conn_get_security`(struct bt_conn *conn)

Get security level for a connection.

Returns Connection security level

uint8_t `bt_conn_enc_key_size`(struct bt_conn *conn)

Get encryption key size.

This function gets encryption key size. If there is no security (encryption) enabled 0 will be returned.

Parameters

- `conn` – Existing connection object.

Returns Encryption key size.

void `bt_conn_cb_register`(struct *bt_conn_cb* *cb)

Register connection callbacks.

Register callbacks to monitor the state of connections.

Parameters

- `cb` – Callback struct. Must point to memory that remains valid.

void `bt_set_bondable`(bool enable)

Enable/disable bonding.

Set/clear the Bonding flag in the Authentication Requirements of SMP Pairing Request/Response data. The initial value of this flag depends on BT_BONDABLE Kconfig setting. For the vast majority of applications calling this function shouldn't be needed.

Parameters

- `enable` – Value allowing/disallowing to be bondable.

void `bt_set_oob_data_flag`(bool enable)

Allow/disallow remote OOB data to be used for pairing.

Set/clear the OOB data flag for SMP Pairing Request/Response data. The initial value of this flag depends on BT_OOB_DATA_PRESENT Kconfig setting.

Parameters

- `enable` – Value allowing/disallowing remote OOB data.

int `bt_le_oob_set_legacy_tk`(struct bt_conn *conn, const uint8_t *tk)

Set OOB Temporary Key to be used for pairing.

This function allows to set OOB data for the LE legacy pairing procedure. The function should only be called in response to the oob_data_request() callback provided that the legacy method is user pairing.

Parameters

- `conn` – Connection object
- `tk` – Pointer to 16 byte long TK array

Returns Zero on success or -EINVAL if NULL

int `bt_le_oob_set_sc_data`(struct bt_conn *conn, const struct *bt_le_oob_sc_data* *oobd_local, const struct *bt_le_oob_sc_data* *oobd_remote)

Set OOB data during LE Secure Connections (SC) pairing procedure.

This function allows to set OOB data during the LE SC pairing procedure. The function should only be called in response to the oob_data_request() callback provided that LE SC method is used for pairing.

The user should submit OOB data according to the information received in the callback. This may yield three different configurations: with only local OOB data present, with only remote OOB data present or with both local and remote OOB data present.

Parameters

- `conn` – Connection object
- `oobd_local` – Local OOB data or NULL if not present
- `oobd_remote` – Remote OOB data or NULL if not present

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int `bt_le_oob_get_sc_data`(struct bt_conn *conn, const struct *bt_le_oob_sc_data* **oobd_local, const struct *bt_le_oob_sc_data* **oobd_remote)

Get OOB data used for LE Secure Connections (SC) pairing procedure.

This function allows to get OOB data during the LE SC pairing procedure that were set by the *bt_le_oob_set_sc_data()* API.

Note: The OOB data will only be available as long as the connection object associated with it is valid.

Parameters

- `conn` – Connection object
- `oobd_local` – Local OOB data or NULL if not set
- `oobd_remote` – Remote OOB data or NULL if not set

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int `bt_passkey_set`(unsigned int passkey)

Set a fixed passkey to be used for pairing.

This API is only available when the CONFIG_BT_FIXED_PASSKEY configuration option has been enabled.

Sets a fixed passkey to be used for pairing. If set, the pairing_confim() callback will be called for all incoming pairings.

Parameters

- `passkey` – A valid passkey (0 - 999999) or BT_PASSKEY_INVALID to disable a previously set fixed passkey.

Returns 0 on success or a negative error code on failure.

int `bt_conn_auth_cb_register`(const struct *bt_conn_auth_cb* *cb)

Register authentication callbacks.

Register callbacks to handle authenticated pairing. Passing NULL unregisters a previous callbacks structure.

Parameters

- `cb` – Callback struct.

Returns Zero on success or negative error code otherwise

int `bt_conn_auth_passkey_entry`(struct bt_conn *conn, unsigned int passkey)

Reply with entered passkey.

This function should be called only after passkey_entry callback from *bt_conn_auth_cb* structure was called.

Parameters

- `conn` – Connection object.
- `passkey` – Entered passkey.

Returns Zero on success or negative error code otherwise

int `bt_conn_auth_cancel`(struct bt_conn *conn)

Cancel ongoing authenticated pairing.

This function allows to cancel ongoing authenticated pairing.

Parameters

- `conn` – Connection object.

Returns Zero on success or negative error code otherwise

int `bt_conn_auth_passkey_confirm`(struct bt_conn *conn)

Reply if passkey was confirmed to match by user.

This function should be called only after passkey_confirm callback from *bt_conn_auth_cb* structure was called.

Parameters

- `conn` – Connection object.

Returns Zero on success or negative error code otherwise

int `bt_conn_auth_pairing_confirm`(struct bt_conn *conn)

Reply if incoming pairing was confirmed by user.

This function should be called only after pairing_confirm callback from *bt_conn_auth_cb* structure was called if user confirmed incoming pairing.

Parameters

- `conn` – Connection object.

Returns Zero on success or negative error code otherwise

int `bt_conn_auth_pincode_entry`(struct bt_conn *conn, const char *pin)

Reply with entered PIN code.

This function should be called only after PIN code callback from *bt_conn_auth_cb* structure was called. It's for legacy 2.0 devices.

Parameters

- `conn` – Connection object.
- `pin` – Entered PIN code.

Returns Zero on success or negative error code otherwise

struct bt_conn *`bt_conn_create_br`(const *bt_addr_t* *peer, const struct *bt_br_conn_param* *param)

Initiate an BR/EDR connection to a remote device.

Allows initiate new BR/EDR link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Parameters

- `peer` – Remote address.
- `param` – Initial connection parameters.

Returns Valid connection object on success or NULL otherwise.

struct bt_conn *`bt_conn_create_sco`(const *bt_addr_t* *peer)

Initiate an SCO connection to a remote device.

Allows initiate new SCO link to remote peer using its address.

The caller gets a new reference to the connection object which must be released with *bt_conn_unref()* once done using the object.

Parameters

- `peer` – Remote address.

Returns Valid connection object on success or NULL otherwise.

struct `bt_le_conn_param`

#include <*conn.h*> Connection parameters for LE connections

struct `bt_conn_le_phy_info`

#include <*conn.h*> Connection PHY information for LE connections

Public Members

uint8_t `rx_phy`

Connection transmit PHY

struct `bt_conn_le_phy_param`

#include <*conn.h*> Preferred PHY parameters for LE connections

Public Members

uint8_t `pref_tx_phy`
Connection PHY options.

uint8_t `pref_rx_phy`
Bitmask of preferred transmit PHYs

struct `bt_conn_le_data_len_info`
#include <conn.h> Connection data length information for LE connections

Public Members

uint16_t `tx_max_len`
Maximum Link Layer transmission payload size in bytes.

uint16_t `tx_max_time`
Maximum Link Layer transmission payload time in us.

uint16_t `rx_max_len`
Maximum Link Layer reception payload size in bytes.

uint16_t `rx_max_time`
Maximum Link Layer reception payload time in us.

struct `bt_conn_le_data_len_param`
#include <conn.h> Connection data length parameters for LE connections

Public Members

uint16_t `tx_max_len`
Maximum Link Layer transmission payload size in bytes.

uint16_t `tx_max_time`
Maximum Link Layer transmission payload time in us.

struct `bt_conn_le_info`
#include <conn.h> LE Connection Info Structure

Public Members

const *bt_addr_le_t* *`src`
Source (Local) Identity Address

const *bt_addr_le_t* *`dst`
Destination (Remote) Identity Address or remote Resolvable Private Address (RPA) before identity has been resolved.

const *bt_addr_le_t* *`local`

Local device address used during connection setup.

const *bt_addr_le_t* *`remote`

Remote device address used during connection setup.

uint16_t `latency`

Connection interval

uint16_t `timeout`

Connection peripheral latency

const struct *bt_conn_le_phy_info* *`phy`

Connection supervision timeout

struct `bt_conn_br_info`

#include <conn.h> BR/EDR Connection Info Structure

struct `bt_conn_info`

#include <conn.h> Connection Info Structure

Public Members

uint8_t `type`

Connection Type.

uint8_t `role`

Connection Role.

uint8_t `id`

Which local identity the connection was created with

struct *bt_conn_le_info* `le`

LE Connection specific Info.

struct *bt_conn_br_info* `br`

BR/EDR Connection specific Info.

union *bt_conn_info*.[anonymous] `[anonymous]`

Connection Type specific Info.

struct `bt_conn_le_remote_info`

#include <conn.h> LE Connection Remote Info Structure

Public Members

const uint8_t *features

Remote LE feature set (bitmask).

struct bt_conn_br_remote_info

#include <conn.h> BR/EDR Connection Remote Info structure

Public Members

const uint8_t *features

Remote feature set (pages of bitmasks).

uint8_t num_pages

Number of pages in the remote feature set.

struct bt_conn_remote_info

#include <conn.h> Connection Remote Info Structure.

Note: The version, manufacturer and subversion fields will only contain valid data if CONFIG_BT_REMOTE_VERSION is enabled.

Public Members

uint8_t type

Connection Type

uint8_t version

Remote Link Layer version

uint16_t manufacturer

Remote manufacturer identifier

uint16_t subversion

Per-manufacturer unique revision

struct *bt_conn_le_remote_info* le

LE connection remote info

struct *bt_conn_br_remote_info* br

BR/EDR connection remote info

struct bt_conn_le_tx_power

#include <conn.h> LE Transmit Power Level Structure

Public Members

uint8_t `phy`

Input: 1M, 2M, Coded S2 or Coded S8

int8_t `current_level`

Output: current transmit power level

int8_t `max_level`

Output: maximum transmit power level

struct `bt_conn_le_create_param`

#include <conn.h>

Public Members

uint32_t `options`

Bit-field of create connection options.

uint16_t `interval`

Scan interval (N * 0.625 ms)

uint16_t `window`

Scan window (N * 0.625 ms)

uint16_t `interval_coded`

Scan interval LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan interval

uint16_t `window_coded`

Scan window LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan window.

uint16_t `timeout`

Connection initiation timeout (N * 10 MS)

Set zero to use the default `CONFIG_BT_CREATE_CONN_TIMEOUT` timeout.

Note: Unused in *bt_conn_le_create_auto*

struct `bt_conn_cb`

#include <conn.h> Connection callback structure.

This structure is used for tracking the state of a connection. It is registered with the help of the *bt_conn_cb_register()* API. It's permissible to register multiple instances of this *bt_conn_cb* type, in case different modules of an application are interested in tracking the connection state. If a callback is not of interest for an instance, it may be set to NULL and will as a consequence not be used for that instance.

Public Members

void (*connected)(struct bt_conn *conn, uint8_t err)

A new connection has been established.

This callback notifies the application of a new connection. In case the err parameter is non-zero it means that the connection establishment failed.

err can mean either of the following:

- BT_HCI_ERR_UNKNOWN_CONN_ID Creating the connection started by *bt_conn_le_create* was canceled either by the user through *bt_conn_disconnect* or by the timeout in the host through *bt_conn_le_create_param* timeout parameter, which defaults to CONFIG_BT_CREATE_CONN_TIMEOUT seconds.
- BT_HCI_ERR_ADV_TIMEOUT High duty cycle directed connectable advertiser started by *bt_le_adv_start* failed to be connected within the timeout.

Note: If the connection was established from an advertising set then the advertising set cannot be restarted directly from this callback. Instead use the connected callback of the advertising set.

Param conn New connection object.
Param err HCI error. Zero for success, non-zero otherwise.

void (*disconnected)(struct bt_conn *conn, uint8_t reason)

A connection has been disconnected.

This callback notifies the application that a connection has been disconnected.

When this callback is called the stack still has one reference to the connection object. If the application in this callback tries to start either a connectable advertiser or create a new connection this might fail because there are no free connection objects available. To avoid this issue it is recommended to either start connectable advertise or create a new connection using *k_work_submit* or increase CONFIG_BT_MAX_CONN .

Param conn Connection object.
Param reason HCI reason for the disconnection.

bool (*le_param_req)(struct bt_conn *conn, struct *bt_le_conn_param* *param)

LE connection parameter update request.

This callback notifies the application that a remote device is requesting to update the connection parameters. The application accepts the parameters by returning true, or rejects them by returning false. Before accepting, the application may also adjust the parameters to better suit its needs.

It is recommended for an application to have just one of these callbacks for simplicity. However, if an application registers multiple it needs to manage the potentially different requirements for each callback. Each callback gets the parameters as returned by previous callbacks, i.e. they are not necessarily the same ones as the remote originally sent.

If the application does not have this callback then the default is to accept the parameters.

Param conn Connection object.
Param param Proposed connection parameters.
Return true to accept the parameters, or false to reject them.

void (*le_param_updated)(struct bt_conn *conn, uint16_t interval, uint16_t latency, uint16_t timeout)

The parameters for an LE connection have been updated.

This callback notifies the application that the connection parameters for an LE connection have been updated.

Param conn Connection object.
Param interval Connection interval.
Param latency Connection latency.
Param timeout Connection supervision timeout.

void (*`identity_resolved`)(struct bt_conn *conn, const *bt_addr_le_t* *rpa, const *bt_addr_le_t* *identity)

Remote Identity Address has been resolved.

This callback notifies the application that a remote Identity Address has been resolved

Param conn Connection object.
Param rpa Resolvable Private Address.
Param identity Identity Address.

void (*`security_changed`)(struct bt_conn *conn, *bt_security_t* level, enum *bt_security_err* err)

The security level of a connection has changed.

This callback notifies the application that the security of a connection has changed.

The security level of the connection can either have been increased or remain unchanged. An increased security level means that the pairing procedure has been performed or the bond information from a previous connection has been applied. If the security level remains unchanged this means that the encryption key has been refreshed for the connection.

Param conn Connection object.
Param level New security level of the connection.
Param err Security error. Zero for success, non-zero otherwise.

void (*`remote_info_available`)(struct bt_conn *conn, struct *bt_conn_remote_info* *remote_info)

Remote information procedures has completed.

This callback notifies the application that the remote information has been retrieved from the remote peer.

Param conn Connection object.
Param remote_info Connection information of remote device.

void (*`le_phy_updated`)(struct bt_conn *conn, struct *bt_conn_le_phy_info* *param)

The PHY of the connection has changed.

This callback notifies the application that the PHY of the connection has changed.

Param conn Connection object.
Param info Connection LE PHY information.

void (*`le_data_len_updated`)(struct bt_conn *conn, struct *bt_conn_le_data_len_info* *info)

The data length parameters of the connection has changed.

This callback notifies the application that the maximum Link Layer payload length or transmission time has changed.

Param conn Connection object.
Param info Connection data length information.

struct `bt_conn_oob_info`

#include <conn.h> Info Structure for OOB pairing

Public Types

enum [anonymous]

Type of OOB pairing method

Values:

enumerator BT_CONN_OOB_LE_LEGACY

LE legacy pairing

enumerator BT_CONN_OOB_LE_SC

LE SC pairing

Public Members

enum *bt_conn_oob_info*.*[anonymous]* type

Type of OOB pairing method

enum *bt_conn_oob_info*.[anonymous].[anonymous].[anonymous] oob_config

OOB data configuration

struct *bt_conn_oob_info*.[anonymous].[anonymous] lesc

LE Secure Connections OOB pairing parameters

struct bt_conn_pairing_feat

#include <conn.h> Pairing request and pairing response info structure.

This structure is the same for both smp_pairing_req and smp_pairing_rsp and a subset of the packet data, except for the initial Code octet. It is documented in Core Spec. Vol. 3, Part H, 3.5.1 and 3.5.2.

Public Members

uint8_t io_capability

IO Capability, Core Spec. Vol 3, Part H, 3.5.1, Table 3.4

uint8_t oob_data_flag

OOB data flag, Core Spec. Vol 3, Part H, 3.5.1, Table 3.5

uint8_t auth_req

AuthReq, Core Spec. Vol 3, Part H, 3.5.1, Fig. 3.3

uint8_t max_enc_key_size

Maximum Encryption Key Size, Core Spec. Vol 3, Part H, 3.5.1

uint8_t init_key_dist

Initiator Key Distribution/Generation, Core Spec. Vol 3, Part H, 3.6.1, Fig. 3.11

uint8_t `resp_key_dist`

Responder Key Distribution/Generation, Core Spec. Vol 3, Part H 3.6.1, Fig. 3.11

struct `bt_conn_auth_cb`

#include <conn.h> Authenticated pairing callback structure

Public Members

enum *bt_security_err* (*`pairing_accept`)(struct bt_conn *conn, const struct *bt_conn_pairing_feat* *const feat)

Query to proceed incoming pairing or not.

On any incoming pairing req/rsp this callback will be called for the application to decide whether to allow for the pairing to continue.

The pairing info received from the peer is passed to assist making the decision.

As this callback is synchronous the application should return a response value immediately. Otherwise it may affect the timing during pairing. Hence, this information should not be conveyed to the user to take action.

The remaining callbacks are not affected by this, but do notice that other callbacks can be called during the pairing. Eg. if pairing_confirm is registered both will be called for Just-Works pairings.

This callback may be unregistered in which case pairing continues as if the Kconfig flag was not set.

This callback is not called for BR/EDR Secure Simple Pairing (SSP).

Param conn Connection where pairing is initiated.
Param feat Pairing req/resp info.

void (*`passkey_display`)(struct bt_conn *conn, unsigned int passkey)

Display a passkey to the user.

When called the application is expected to display the given passkey to the user, with the expectation that the passkey will then be entered on the peer device. The passkey will be in the range of 0 - 999999, and is expected to be padded with zeroes so that six digits are always shown. E.g. the value 37 should be shown as 000037.

This callback may be set to NULL, which means that the local device lacks the ability do display a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop displaying the passkey.

Param conn Connection where pairing is currently active.
Param passkey Passkey to show to the user.

void (*`passkey_entry`)(struct bt_conn *conn)

Request the user to enter a passkey.

When called the user is expected to enter a passkey. The passkey must be in the range of 0 - 999999, and should be expected to be zero-padded, as that's how the peer device will typically be showing it (e.g. 37 would be shown as 000037).

Once the user has entered the passkey its value should be given to the stack using the *bt_conn_auth_passkey_entry()* API.

This callback may be set to NULL, which means that the local device lacks the ability to enter a passkey. If set to non-NULL the cancel callback must also be provided, since this is

the only way the application can find out that it should stop requesting the user to enter a passkey.

Param conn Connection where pairing is currently active.

void (*passkey_confirm)(struct bt_conn *conn, unsigned int passkey)

Request the user to confirm a passkey.

When called the user is expected to confirm that the given passkey is also shown on the peer device.. The passkey will be in the range of 0 - 999999, and should be zero-padded to always be six digits (e.g. 37 would be shown as 000037).

Once the user has confirmed the passkey to match, the *bt_conn_auth_passkey_confirm()* API should be called. If the user concluded that the passkey doesn't match the *bt_conn_auth_cancel()* API should be called.

This callback may be set to NULL, which means that the local device lacks the ability to confirm a passkey. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to confirm a passkey.

Param conn Connection where pairing is currently active.

Param passkey Passkey to be confirmed.

void (*oob_data_request)(struct bt_conn *conn, struct *bt_conn_oob_info* *info)

Request the user to provide Out of Band (OOB) data.

When called the user is expected to provide OOB data. The required data are indicated by the information structure.

For LE Secure Connections OOB pairing, the user should provide local OOB data, remote OOB data or both depending on their availability. Their value should be given to the stack using the *bt_le_oob_set_sc_data()* API.

This callback must be set to non-NULL in order to support OOB pairing.

Param conn Connection where pairing is currently active.

Param info OOB pairing information.

void (*cancel)(struct bt_conn *conn)

Cancel the ongoing user request.

This callback will be called to notify the application that it should cancel any previous user request (passkey display, entry or confirmation).

This may be set to NULL, but must always be provided whenever the passkey_display, passkey_entry passkey_confirm or pairing_confirm callback has been provided.

Param conn Connection where pairing is currently active.

void (*pairing_confirm)(struct bt_conn *conn)

Request confirmation for an incoming pairing.

This callback will be called to confirm an incoming pairing request where none of the other user callbacks is applicable.

If the user decides to accept the pairing the *bt_conn_auth_pairing_confirm()* API should be called. If the user decides to reject the pairing the *bt_conn_auth_cancel()* API should be called.

This callback may be set to NULL, which means that the local device lacks the ability to confirm a pairing request. If set to non-NULL the cancel callback must also be provided, since this is the only way the application can find out that it should stop requesting the user to confirm a pairing request.

Param conn Connection where pairing is currently active.

void (*pincode_entry)(struct bt_conn *conn, bool highsec)

Request the user to enter a passkey.

This callback will be called for a BR/EDR (Bluetooth Classic) connection where pairing is being performed. Once called the user is expected to enter a PIN code with a length between 1 and 16 digits. If the *highsec* parameter is set to true the PIN code must be 16 digits long.

Once entered, the PIN code should be given to the stack using the *bt_conn_auth_pincode_entry()* API.

This callback may be set to NULL, however in that case pairing over BR/EDR will not be possible. If provided, the cancel callback must be provided as well.

Param conn Connection where pairing is currently active.
Param highsec true if 16 digit PIN is required.

void (*pairing_complete)(struct bt_conn *conn, bool bonded)

notify that pairing procedure was complete.

This callback notifies the application that the pairing procedure has been completed.

Param conn Connection object.
Param bonded Bond information has been distributed during the pairing procedure.

void (*pairing_failed)(struct bt_conn *conn, enum *bt_security_err* reason)

notify that pairing process has failed.

Param conn Connection object.
Param reason Pairing failed reason

void (*bond_deleted)(uint8_t id, const *bt_addr_le_t* *peer)

Notify that bond has been deleted.

This callback notifies the application that the bond information for the remote peer has been deleted

Param id Which local identity had the bond.
Param peer Remote address.

struct bt_br_conn_param

#include <conn.h> Connection parameters for BR/EDR connections

7.4.2 Bluetooth Controller

API Reference

group bt_ctrl

Bluetooth Controller.

Functions

void bt_ctlr_set_public_addr(const uint8_t *addr)

Set public address for controller.

Should be called before *bt_enable()*.

Parameters

- `addr` – Public address

7.4.3 Cryptography

API Reference

group `bt_crypto`

Cryptography.

Functions

int `bt_rand`(void *buf, size_t len)

Generate random data.

A random number generation helper which utilizes the Bluetooth controller's own RNG.

Parameters

- `buf` – Buffer to insert the random data
- `len` – Length of random data to generate

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error

int `bt_encrypt_le`(const uint8_t key[16], const uint8_t plaintext[16], uint8_t enc_data[16])

AES encrypt little-endian data.

An AES encrypt helper is used to request the Bluetooth controller's own hardware to encrypt the plaintext using the key and returns the encrypted data.

Parameters

- `key` – 128 bit LS byte first key for the encryption of the plaintext
- `plaintext` – 128 bit LS byte first plaintext data block to be encrypted
- `enc_data` – 128 bit LS byte first encrypted data block

Returns Zero on success or error code otherwise.

int `bt_encrypt_be`(const uint8_t key[16], const uint8_t plaintext[16], uint8_t enc_data[16])

AES encrypt big-endian data.

An AES encrypt helper is used to request the Bluetooth controller's own hardware to encrypt the plaintext using the key and returns the encrypted data.

Parameters

- `key` – 128 bit MS byte first key for the encryption of the plaintext
- `plaintext` – 128 bit MS byte first plaintext data block to be encrypted
- `enc_data` – 128 bit MS byte first encrypted data block

Returns Zero on success or error code otherwise.

int `bt_ccm_decrypt`(const uint8_t key[16], uint8_t nonce[13], const uint8_t *enc_data, size_t len, const uint8_t *aad, size_t aad_len, uint8_t *plaintext, size_t mic_size)

Decrypt big-endian data with AES-CCM.

Decrypts and authorizes `enc_data` with AES-CCM, as described in https://tools.ietf.org/html/rfc3610.

Assumes that the MIC follows directly after the encrypted data.

Parameters

- `key` – 128 bit MS byte first key
- `nonce` – 13 byte MS byte first nonce
- `enc_data` – Encrypted data
- `len` – Length of the encrypted data
- `aad` – Additional input data
- `aad_len` – Additional input data length
- `plaintext` – Plaintext buffer to place result in
- `mic_size` – Size of the trailing MIC (in bytes)

Return values

- `0` – Successfully decrypted the data.
- `-EINVAL` – Invalid parameters.
- `-EBADMSG` – Authentication failed.

int `bt_ccm_encrypt`(const uint8_t key[16], uint8_t nonce[13], const uint8_t *plaintext, size_t len, const uint8_t *aad, size_t aad_len, uint8_t *enc_data, size_t mic_size)

Encrypt big-endian data with AES-CCM.

Encrypts and generates a MIC from `plaintext` with AES-CCM, as described in https://tools.ietf.org/html/rfc3610.

Places the MIC directly after the encrypted data.

Parameters

- `key` – 128 bit MS byte first key
- `nonce` – 13 byte MS byte first nonce
- `plaintext` – Plaintext buffer to encrypt
- `len` – Length of the encrypted data
- `aad` – Additional input data
- `aad_len` – Additional input data length
- `enc_data` – Buffer to place encrypted data in
- `mic_size` – Size of the trailing MIC (in bytes)

Return values

- `0` – Successfully encrypted the data.
- `-EINVAL` – Invalid parameters.

7.4.4 Data Buffers

API Reference

group `bt_buf`

Data buffers.

Defines

BT_BUF_RESERVE

BT_BUF_SIZE(size)
Helper to include reserved HCI data in buffer calculations

BT_BUF_ACL_SIZE(size)
Helper to calculate needed buffer size for HCI ACL packets

BT_BUF_EVT_SIZE(size)
Helper to calculate needed buffer size for HCI Event packets.

BT_BUF_CMD_SIZE(size)
Helper to calculate needed buffer size for HCI Command packets.

BT_BUF_ACL_RX_SIZE
Data size needed for HCI ACL RX buffers

BT_BUF_EVT_RX_SIZE
Data size needed for HCI Event RX buffers

BT_BUF_RX_SIZE
Data size needed for HCI ACL or Event RX buffers

BT_BUF_CMD_TX_SIZE
Data size needed for HCI Command buffers.

Enums

enum bt_buf_type
Possible types of buffers passed around the Bluetooth stack

Values:

enumerator BT_BUF_CMD
HCI command

enumerator BT_BUF_EVT
HCI event

enumerator BT_BUF_ACL_OUT
Outgoing ACL data

enumerator BT_BUF_ACL_IN
Incoming ACL data

enumerator BT_BUF_ISO_OUT
Outgoing ISO data

enumerator BT_BUF_ISO_IN

Incoming ISO data

enumerator BT_BUF_H4

H:4 data

Functions

struct *net_buf* *bt_buf_get_rx(enum *bt_buf_type* type, *k_timeout_t* timeout)

Allocate a buffer for incoming data

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before *bt_recv_prio()*.

Parameters

- type – Type of buffer. Only BT_BUF_EVT and BT_BUF_ACL_IN are allowed.
- timeout – Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

Returns A new buffer.

struct *net_buf* *bt_buf_get_tx(enum *bt_buf_type* type, *k_timeout_t* timeout, const void *data, size_t size)

Allocate a buffer for outgoing data

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before *bt_send()*.

Parameters

- type – Type of buffer. Only BT_BUF_CMD, BT_BUF_ACL_OUT or BT_BUF_H4, when operating on H:4 mode, are allowed.
- timeout – Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.
- data – Initial data to append to buffer.
- size – Initial data size.

Returns A new buffer.

struct *net_buf* *bt_buf_get_cmd_complete(*k_timeout_t* timeout)

Allocate a buffer for an HCI Command Complete/Status Event

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before *bt_recv_prio()*.

Parameters

- timeout – Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

Returns A new buffer.

struct *net_buf* *bt_buf_get_evt(uint8_t evt, bool discardable, *k_timeout_t* timeout)

Allocate a buffer for an HCI Event

This will set the buffer type so *bt_buf_set_type()* does not need to be explicitly called before *bt_recv_prio()* or *bt_recv()*.

Parameters

- evt – HCI event code

- `discardable` – Whether the driver considers the event discardable.
- `timeout` – Non-negative waiting period to obtain a buffer or one of the special values K_NO_WAIT and K_FOREVER.

Returns A new buffer.

static inline void `bt_buf_set_type`(struct *net_buf* *buf, enum *bt_buf_type* type)

Set the buffer type

Parameters

- `buf` – Bluetooth buffer
- `type` – The BT_* type to set the buffer to

static inline enum *bt_buf_type* `bt_buf_get_type`(struct *net_buf* *buf)

Get the buffer type

Parameters

- `buf` – Bluetooth buffer

Returns The BT_* type to of the buffer

struct `bt_buf_data`

#include <buf.h> This is a base type for bt_buf user data.

7.4.5 Generic Access Profile (GAP)

API Reference

group `bt_gap`

Generic Access Profile.

Defines

`BT_ID_DEFAULT`

Convenience macro for specifying the default identity. This helps make the code more readable, especially when only one identity is supported.

`BT_DATA`(_type, _data, _data_len)

Helper to declare elements of *bt_data* arrays.

This macro is mainly for creating an array of struct *bt_data* elements which is then passed to e.g. *bt_le_adv_start()*.

Parameters

- `_type` – Type of advertising data field
- `_data` – Pointer to the data field payload
- `_data_len` – Number of bytes behind the _data pointer

`BT_DATA_BYTES`(_type, _bytes...)

Helper to declare elements of *bt_data* arrays.

This macro is mainly for creating an array of struct *bt_data* elements which is then passed to e.g. *bt_le_adv_start()*.

Parameters

- _type – Type of advertising data field
- _bytes – Variable number of single-byte parameters

BT_LE_ADV_PARAM_INIT(_options, _int_min, _int_max, _peer)

Initialize advertising parameters.

Parameters

- _options – Advertising Options
- _int_min – Minimum advertising interval
- _int_max – Maximum advertising interval
- _peer – Peer address, set to NULL for undirected advertising or address of peer for directed advertising.

BT_LE_ADV_PARAM(_options, _int_min, _int_max, _peer)

Helper to declare advertising parameters inline.

Parameters

- _options – Advertising Options
- _int_min – Minimum advertising interval
- _int_max – Maximum advertising interval
- _peer – Peer address, set to NULL for undirected advertising or address of peer for directed advertising.

BT_LE_ADV_CONN_DIR(_peer)

BT_LE_ADV_CONN

BT_LE_ADV_CONN_NAME

BT_LE_ADV_CONN_NAME_AD

BT_LE_ADV_CONN_DIR_LOW_DUTY(_peer)

BT_LE_ADV_NCONN

Non-connectable advertising with private address

BT_LE_ADV_NCONN_NAME

Non-connectable advertising with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_ADV_NCONN_IDENTITY

Non-connectable advertising with *BT_LE_ADV_OPT_USE_IDENTITY*

BT_LE_EXT_ADV_CONN_NAME

Connectable extended advertising with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_EXT_ADV_SCAN_NAME

Scannable extended advertising with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_EXT_ADV_NCONN

Non-connectable extended advertising with private address

BT_LE_EXT_ADV_NCONN_NAME

Non-connectable extended advertising with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_EXT_ADV_NCONN_IDENTITY

Non-connectable extended advertising with *BT_LE_ADV_OPT_USE_IDENTITY*

BT_LE_EXT_ADV_CODED_NCONN

Non-connectable extended advertising on coded PHY with private address

BT_LE_EXT_ADV_CODED_NCONN_NAME

Non-connectable extended advertising on coded PHY with *BT_LE_ADV_OPT_USE_NAME*

BT_LE_EXT_ADV_CODED_NCONN_IDENTITY

Non-connectable extended advertising on coded PHY with *BT_LE_ADV_OPT_USE_IDENTITY*

BT_LE_EXT_ADV_START_PARAM_INIT(_timeout, _n_evts)

Helper to initialize extended advertising start parameters inline

Parameters

- _timeout – Advertiser timeout
- _n_evts – Number of advertising events

BT_LE_EXT_ADV_START_PARAM(_timeout, _n_evts)

Helper to declare extended advertising start parameters inline

Parameters

- _timeout – Advertiser timeout
- _n_evts – Number of advertising events

BT_LE_EXT_ADV_START_DEFAULT

BT_LE_PER_ADV_PARAM_INIT(_int_min, _int_max, _options)

Helper to declare periodic advertising parameters inline

Parameters

- _int_min – Minimum periodic advertising interval
- _int_max – Maximum periodic advertising interval
- _options – Periodic advertising properties bitfield.

BT_LE_PER_ADV_PARAM(_int_min, _int_max, _options)

Helper to declare periodic advertising parameters inline

Parameters

- _int_min – Minimum periodic advertising interval
- _int_max – Maximum periodic advertising interval
- _options – Periodic advertising properties bitfield.

BT_LE_PER_ADV_DEFAULT

BT_LE_SCAN_OPT_FILTER_WHITELIST

BT_LE_SCAN_PARAM_INIT(_type, _options, _interval, _window)

Initialize scan parameters.

Parameters

- _type – Scan Type, BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE.
- _options – Scan options
- _interval – Scan Interval (N * 0.625 ms)
- _window – Scan Window (N * 0.625 ms)

BT_LE_SCAN_PARAM(_type, _options, _interval, _window)

Helper to declare scan parameters inline.

Parameters

- _type – Scan Type, BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE.
- _options – Scan options
- _interval – Scan Interval (N * 0.625 ms)
- _window – Scan Window (N * 0.625 ms)

BT_LE_SCAN_ACTIVE

Helper macro to enable active scanning to discover new devices.

BT_LE_SCAN_PASSIVE

Helper macro to enable passive scanning to discover new devices.

This macro should be used if information required for device identification (e.g., UUID) are known to be placed in Advertising Data.

BT_LE_SCAN_CODED_ACTIVE

Helper macro to enable active scanning to discover new devices. Include scanning on Coded PHY in addition to 1M PHY.

BT_LE_SCAN_CODED_PASSIVE

Helper macro to enable passive scanning to discover new devices. Include scanning on Coded PHY in addition to 1M PHY.

This macro should be used if information required for device identification (e.g., UUID) are known to be placed in Advertising Data.

Typedefs

typedef void (*bt_ready_cb_t)(int err)

Callback for notifying that Bluetooth has been enabled.

Param err zero on success or (negative) error code otherwise.

typedef void bt_le_scan_cb_t(const *bt_addr_le_t* *addr, int8_t rssi, uint8_t adv_type, struct *net_buf_simple* *buf)

Callback type for reporting LE scan results.

A function of this type is given to the *bt_le_scan_start()* function and will be called for any discovered LE device.

Param addr Advertiser LE address and type.

Param rssi Strength of advertiser signal.

Param adv_type Type of advertising response from advertiser.

Param buf Buffer containing advertiser data.

typedef void bt_br_discovery_cb_t(struct *bt_br_discovery_result* *results, size_t count)

Callback type for reporting BR/EDR discovery (inquiry) results.

A callback of this type is given to the *bt_br_discovery_start()* function and will be called at the end of the discovery with information about found devices populated in the results array.

Param results Storage used for discovery results

Param count Number of valid discovery results.

Enums

enum [anonymous]

Advertising options

Values:

enumerator BT_LE_ADV_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_LE_ADV_OPT_CONNECTABLE = *BIT*(0)

Advertise as connectable.

Advertise as connectable. If not connectable then the type of advertising is determined by providing scan response data. The advertiser address is determined by the type of advertising and/or enabling privacy CONFIG_BT_PRIVACY .

enumerator BT_LE_ADV_OPT_ONE_TIME = *BIT*(1)

Advertise one time.

Don't try to resume connectable advertising after a connection. This option is only meaningful when used together with BT_LE_ADV_OPT_CONNECTABLE. If set the advertising will be stopped when *bt_le_adv_stop()* is called or when an incoming (peripheral) connection happens. If this option is not set the stack will take care of keeping advertising enabled even as connections occur. If Advertising directed or the advertiser was started with *bt_le_ext_adv_start* then this behavior is the default behavior and this flag has no effect.

enumerator BT_LE_ADV_OPT_USE_IDENTITY = *BIT*(2)

Advertise using identity address.

Advertise using the identity address as the advertiser address.

Note: The address used for advertising will not be the same as returned by *bt_le_oob_get_local*, instead *bt_id_get* should be used to get the LE address.

Warning: This will compromise the privacy of the device, so care must be taken when using this option.

enumerator BT_LE_ADV_OPT_USE_NAME = *BIT*(3)

Advertise using GAP device name.

```
Include the GAP device name automatically when advertising.
By default the GAP device name is put at the end of the scan
response data.
When advertising using @ref BT_LE_ADV_OPT_EXT_ADV and not
@ref BT_LE_ADV_OPT_SCANNABLE then it will be put at the end of the
advertising data.
If the GAP device name does not fit into advertising data it will be
converted to a shortened name if possible.
@ref BT_LE_ADV_OPT_FORCE_NAME_IN_AD can be used to force the device
name to appear in the advertising data of an advert with scan
response data.

The application can set the device name itself by including the
following in the advertising data.
@code
BT_DATA(BT_DATA_NAME_COMPLETE, name, sizeof(name) - 1)
@endcode
```

enumerator BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY = *BIT*(4)

Low duty cycle directed advertising.

Use low duty directed advertising mode, otherwise high duty mode will be used.

enumerator BT_LE_ADV_OPT_DIR_ADDR_RPA = *BIT*(5)

Directed advertising to privacy-enabled peer.

Enable use of Resolvable Private Address (RPA) as the target address in directed advertisements. This is required if the remote device is privacy-enabled and supports address resolution of the target address in directed advertisement. It is the responsibility of the application to check that the remote device supports address resolution of directed advertisements by reading its Central Address Resolution characteristic.

enumerator BT_LE_ADV_OPT_FILTER_SCAN_REQ = *BIT*(6)

Use filter accept list to filter devices that can request scan response data.

enumerator BT_LE_ADV_OPT_FILTER_CONN = *BIT*(7)

Use filter accept list to filter devices that can connect.

enumerator BT_LE_ADV_OPT_NOTIFY_SCAN_REQ = *BIT*(8)

Notify the application when a scan response data has been sent to an active scanner.

enumerator BT_LE_ADV_OPT_SCANNABLE = *BIT*(9)

Support scan response data.

When used together with *BT_LE_ADV_OPT_EXT_ADV* then this option cannot be used together with the *BT_LE_ADV_OPT_CONNECTABLE* option. When used together with *BT_LE_ADV_OPT_EXT_ADV* then scan response data must be set.

enumerator BT_LE_ADV_OPT_EXT_ADV = *BIT*(10)

Advertise with extended advertising.

This options enables extended advertising in the advertising set. In extended advertising the advertising set will send a small header packet on the three primary advertising channels. This small header points to the advertising data packet that will be sent on one of the 37 secondary advertising channels. The advertiser will send primary advertising on LE 1M PHY, and secondary advertising on LE 2M PHY. Connections will be established on LE 2M PHY.

Without this option the advertiser will send advertising data on the three primary advertising channels.

Note: Enabling this option requires extended advertising support in the peer devices scanning for advertisement packets.

enumerator BT_LE_ADV_OPT_NO_2M = *BIT*(11)

Disable use of LE 2M PHY on the secondary advertising channel.

Disabling the use of LE 2M PHY could be necessary if scanners don't support the LE 2M PHY. The advertiser will send primary advertising on LE 1M PHY, and secondary advertising on LE 1M PHY. Connections will be established on LE 1M PHY.

Note: Cannot be set if BT_LE_ADV_OPT_CODED is set.

Note: Requires *BT_LE_ADV_OPT_EXT_ADV*.

enumerator BT_LE_ADV_OPT_CODED = *BIT*(12)

Advertise on the LE Coded PHY (Long Range).

The advertiser will send both primary and secondary advertising on the LE Coded PHY. This gives the advertiser increased range with the trade-off of lower data rate and higher power consumption. Connections will be established on LE Coded PHY.

Note: Requires *BT_LE_ADV_OPT_EXT_ADV*

enumerator BT_LE_ADV_OPT_ANONYMOUS = *BIT*(13)

Advertise without a device address (identity or RPA).

Note: Requires *BT_LE_ADV_OPT_EXT_ADV*

enumerator BT_LE_ADV_OPT_USE_TX_POWER = *BIT*(14)

Advertise with transmit power.

Note: Requires *BT_LE_ADV_OPT_EXT_ADV*

enumerator BT_LE_ADV_OPT_DISABLE_CHAN_37 = *BIT*(15)

Disable advertising on channel index 37.

enumerator BT_LE_ADV_OPT_DISABLE_CHAN_38 = *BIT*(16)

Disable advertising on channel index 38.

enumerator BT_LE_ADV_OPT_DISABLE_CHAN_39 = *BIT*(17)

Disable advertising on channel index 39.

enumerator BT_LE_ADV_OPT_FORCE_NAME_IN_AD = *BIT*(18)

Put GAP device name into advert data.

Will place the GAP device name into the advertising data rather than the scan response data.

Note: Requires *BT_LE_ADV_OPT_USE_NAME*

---

enum [anonymous]

Periodic Advertising options

Values:

enumerator BT_LE_PER_ADV_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_LE_PER_ADV_OPT_USE_TX_POWER = *BIT*(1)

Advertise with transmit power.

Note: Requires *BT_LE_ADV_OPT_EXT_ADV*

---

enum [anonymous]

Periodic advertising sync options

Values:

enumerator BT_LE_PER_ADV_SYNC_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_LE_PER_ADV_SYNC_OPT_USE_PER_ADV_LIST = *BIT*(0)

Use the periodic advertising list to sync with advertiser.

When this option is set, the address and SID of the parameters are ignored.

enumerator BT_LE_PER_ADV_SYNC_OPT_REPORTING_INITIALLY_DISABLED = *BIT*(1)

Disables periodic advertising reports.

No advertisement reports will be handled until enabled.

enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOA = *BIT*(2)

Sync with Angle of Arrival (AoA) constant tone extension

enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_1US = *BIT*(3)

Sync with Angle of Departure (AoD) 1 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_OPT_DONT_SYNC_AOD_2US = *BIT*(4)

Sync with Angle of Departure (AoD) 2 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_OPT_SYNC_ONLY_CONST_TONE_EXT = *BIT*(5)

Do not sync to packets without a constant tone extension

enum [anonymous]

Periodic Advertising Sync Transfer options

Values:

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOA = *BIT*(0)

No Angle of Arrival (AoA)

Do not sync with Angle of Arrival (AoA) constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_1US = *BIT*(1)

No Angle of Departure (AoD) 1 us.

Do not sync with Angle of Departure (AoD) 1 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_NO_AOD_2US = *BIT*(2)

No Angle of Departure (AoD) 2.

Do not sync with Angle of Departure (AoD) 2 us constant tone extension

enumerator BT_LE_PER_ADV_SYNC_TRANSFER_OPT_SYNC_ONLY_CTE = *BIT*(3)

Only sync to packets with constant tone extension

enum [anonymous]

Values:

enumerator BT_LE_SCAN_OPT_NONE = 0

Convenience value when no options are specified.

enumerator BT_LE_SCAN_OPT_FILTER_DUPLICATE = *BIT*(0)

Filter duplicates.

enumerator BT_LE_SCAN_OPT_FILTER_ACCEPT_LIST = *BIT*(1)

Filter using filter accept list.

enumerator BT_LE_SCAN_OPT_CODED = *BIT*(2)

Enable scan on coded PHY (Long Range).

enumerator BT_LE_SCAN_OPT_NO_1M = *BIT*(3)

Disable scan on 1M phy.

Note: Requires *BT_LE_SCAN_OPT_CODED*.

enum [anonymous]

Values:

enumerator BT_LE_SCAN_TYPE_PASSIVE = 0x00

Scan without requesting additional information from advertisers.

enumerator BT_LE_SCAN_TYPE_ACTIVE = 0x01

Scan and request additional information from advertisers.

Functions

int bt_enable(*bt_ready_cb_t* cb)

Enable Bluetooth.

Enable Bluetooth. Must be the called before any calls that require communication with the local Bluetooth hardware.

When CONFIG_BT_SETTINGS has been enabled and the application is not managing identities of the stack itself then the application must call *settings_load()* before the stack is fully enabled. See *bt_id_create()* for more information.

Parameters

- cb – Callback to notify completion or NULL to perform the enabling synchronously.

Returns Zero on success or (negative) error code otherwise.

int bt_set_name(const char *name)

Set Bluetooth Device Name.

Set Bluetooth GAP Device Name.

When advertising with device name in the advertising data the name should be updated by calling *bt_le_adv_update_data* or *bt_le_ext_adv_set_data*.

Parameters

- name – New name

Returns Zero on success or (negative) error code otherwise.

const char *bt_get_name(void)

Get Bluetooth Device Name.

Get Bluetooth GAP Device Name.

Returns Bluetooth Device Name

void bt_id_get(*bt_addr_le_t* *addrs, size_t *count)

Get the currently configured identities.

Returns an array of the currently configured identity addresses. To make sure all available identities can be retrieved, the number of elements in the *addrs* array should be CONFIG_BT_ID_MAX. The identity identifier that some APIs expect (such as advertising parameters) is simply the index of the identity in the *addrs* array.

If *addrs* is passed as NULL, then returned *count* contains the count of all available identities that can be retrieved with a subsequent call to this function with non-NULL *addrs* parameter.

Note: Deleted identities may show up as BT_LE_ADDR_ANY in the returned array.

Parameters

- addrs – Array where to store the configured identities.

- `count` – Should be initialized to the array size. Once the function returns it will contain the number of returned identities.

int `bt_id_create`(*bt_addr_le_t* *addr, uint8_t *irk)

Create a new identity.

Create a new identity using the given address and IRK. This function can be called before calling *bt_enable()*, in which case it can be used to override the controller's public address (in case it has one). However, the new identity will only be stored persistently in flash when this API is used after *bt_enable()*. The reason is that the persistent settings are loaded after *bt_enable()* and would therefore cause potential conflicts with the stack blindly overwriting what's stored in flash. The identity will also not be written to flash in case a pre-defined address is provided, since in such a situation the app clearly has some place it got the address from and will be able to repeat the procedure on every power cycle, i.e. it would be redundant to also store the information in flash.

Generating random static address or random IRK is not supported when calling this function before *bt_enable()*.

If the application wants to have the stack randomly generate identities and store them in flash for later recovery, the way to do it would be to first initialize the stack (using bt_enable), then call *settings_load()*, and after that check with *bt_id_get()* how many identities were recovered. If an insufficient amount of identities were recovered the app may then call *bt_id_create()* to create new ones.

Parameters

- `addr` – Address to use for the new identity. If NULL or initialized to BT_ADDR_LE_ANY the stack will generate a new random static address for the identity and copy it to the given parameter upon return from this function (in case the parameter was non-NULL).
- `irk` – Identity Resolving Key (16 bytes) to be used with this identity. If set to all zeroes or NULL, the stack will generate a random IRK for the identity and copy it back to the parameter upon return from this function (in case the parameter was non-NULL). If privacy `CONFIG_BT_PRIVACY` is not enabled this parameter must be NULL.

Returns Identity identifier (>= 0) in case of success, or a negative error code on failure.

int `bt_id_reset`(uint8_t id, *bt_addr_le_t* *addr, uint8_t *irk)

Reset/reclaim an identity for reuse.

The semantics of the *addr* and *irk* parameters of this function are the same as with *bt_id_create()*. The difference is the first *id* parameter that needs to be an existing identity (if it doesn't exist this function will return an error). When given an existing identity this function will disconnect any connections created using it, remove any pairing keys or other data associated with it, and then create a new identity in the same slot, based on the *addr* and *irk* parameters.

Note: the default identity (BT_ID_DEFAULT) cannot be reset, i.e. this API will return an error if asked to do that.

Parameters

- `id` – Existing identity identifier.
- `addr` – Address to use for the new identity. If NULL or initialized to BT_ADDR_LE_ANY the stack will generate a new static random address for the identity and copy it to the given parameter upon return from this function (in case the parameter was non-NULL).

- `irk` – Identity Resolving Key (16 bytes) to be used with this identity. If set to all zeroes or NULL, the stack will generate a random IRK for the identity and copy it back to the parameter upon return from this function (in case the parameter was non-NULL). If privacy `CONFIG_BT_PRIVACY` is not enabled this parameter must be NULL.

Returns Identity identifier (>= 0) in case of success, or a negative error code on failure.

int `bt_id_delete`(uint8_t id)

Delete an identity.

When given a valid identity this function will disconnect any connections created using it, remove any pairing keys or other data associated with it, and then flag is as deleted, so that it can not be used for any operations. To take back into use the slot the identity was occupying the *bt_id_reset()* API needs to be used.

Note: the default identity (BT_ID_DEFAULT) cannot be deleted, i.e. this API will return an error if asked to do that.

Parameters

- `id` – Existing identity identifier.

Returns 0 in case of success, or a negative error code on failure.

int `bt_le_adv_start`(const struct *bt_le_adv_param* *param, const struct *bt_data* *ad, size_t ad_len, const struct *bt_data* *sd, size_t sd_len)

Start advertising.

Set advertisement data, scan response data, advertisement parameters and start advertising.

When the advertisement parameter peer address has been set the advertising will be directed to the peer. In this case advertisement data and scan response data parameters are ignored. If the mode is high duty cycle the timeout will be *BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT*.

Parameters

- `param` – Advertising parameters.
- `ad` – Data to be used in advertisement packets.
- `ad_len` – Number of elements in ad
- `sd` – Data to be used in scan response packets.
- `sd_len` – Number of elements in sd

Returns Zero on success or (negative) error code otherwise.

Returns -ENOMEM No free connection objects available for connectable advertiser.

Returns -ECONNREFUSED When connectable advertising is requested and there is already maximum number of connections established in the controller. This error code is only guaranteed when using Zephyr controller, for other controllers code returned in this case may be -EIO.

int `bt_le_adv_update_data`(const struct *bt_data* *ad, size_t ad_len, const struct *bt_data* *sd, size_t sd_len)

Update advertising.

Update advertisement and scan response data.

Parameters

- `ad` – Data to be used in advertisement packets.
- `ad_len` – Number of elements in ad
- `sd` – Data to be used in scan response packets.
- `sd_len` – Number of elements in sd

Returns Zero on success or (negative) error code otherwise.

int `bt_le_adv_stop`(void)

Stop advertising.

Stops ongoing advertising.

Returns Zero on success or (negative) error code otherwise.

int `bt_le_ext_adv_create`(const struct *bt_le_adv_param* *param, const struct *bt_le_ext_adv_cb* *cb, struct bt_le_ext_adv **adv)

Create advertising set.

Create a new advertising set and set advertising parameters. Advertising parameters can be updated with *bt_le_ext_adv_update_param*.

Parameters

- `param` – **[in]** Advertising parameters.
- `cb` – **[in]** Callback struct to notify about advertiser activity. Can be NULL. Must point to valid memory during the lifetime of the advertising set.
- `adv` – **[out]** Valid advertising set object on success.

Returns Zero on success or (negative) error code otherwise.

int `bt_le_ext_adv_start`(struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_start_param* *param)

Start advertising with the given advertising set.

If the advertiser is limited by either the timeout or number of advertising events the application will be notified by the advertiser sent callback once the limit is reached. If the advertiser is limited by both the timeout and the number of advertising events then the limit that is reached first will stop the advertiser.

Parameters

- `adv` – Advertising set object.
- `param` – Advertise start parameters.

int `bt_le_ext_adv_stop`(struct bt_le_ext_adv *adv)

Stop advertising with the given advertising set.

Stop advertising with a specific advertising set. When using this function the advertising sent callback will not be called.

Parameters

- `adv` – Advertising set object.

Returns Zero on success or (negative) error code otherwise.

int `bt_le_ext_adv_set_data`(struct bt_le_ext_adv *adv, const struct *bt_data* *ad, size_t ad_len, const struct *bt_data* *sd, size_t sd_len)

Set an advertising set's advertising or scan response data.

Set advertisement data or scan response data. If the advertising set is currently advertising then the advertising data will be updated in subsequent advertising events.

When both *BT_LE_ADV_OPT_EXT_ADV* and *BT_LE_ADV_OPT_SCANNABLE* are enabled then advertising data is ignored. When *BT_LE_ADV_OPT_SCANNABLE* is not enabled then scan response data is ignored.

If the advertising set has been configured to send advertising data on the primary advertising channels then the maximum data length is *BT_GAP_ADV_MAX_ADV_DATA_LEN* bytes. If the advertising set has been configured for extended advertising, then the maximum data length is defined by the controller with the maximum possible of *BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN* bytes.

Note: Not all scanners support extended data length advertising data.

Note: When updating the advertising data while advertising the advertising data and scan response data length must be smaller or equal to what can be fit in a single advertising packet. Otherwise the advertiser must be stopped.

Parameters

- `adv` – Advertising set object.
- `ad` – Data to be used in advertisement packets.
- `ad_len` – Number of elements in ad
- `sd` – Data to be used in scan response packets.
- `sd_len` – Number of elements in sd

Returns Zero on success or (negative) error code otherwise.

int `bt_le_ext_adv_update_param`(struct bt_le_ext_adv *adv, const struct *bt_le_adv_param* *param)

Update advertising parameters.

Update the advertising parameters. The function will return an error if the advertiser set is currently advertising. Stop the advertising set before calling this function.

Note: When changing the option *BT_LE_ADV_OPT_USE_NAME* then *bt_le_ext_adv_set_data* needs to be called in order to update the advertising data and scan response data.

Parameters

- `adv` – Advertising set object.
- `param` – Advertising parameters.

Returns Zero on success or (negative) error code otherwise.

int `bt_le_ext_adv_delete`(struct bt_le_ext_adv *adv)

Delete advertising set.

Delete advertising set. This will free up the advertising set and make it possible to create a new advertising set.

Returns Zero on success or (negative) error code otherwise.

uint8_t bt_le_ext_adv_get_index(struct bt_le_ext_adv *adv)

Get array index of an advertising set.

This function is used to map bt_adv to index of an array of advertising sets. The array has CONFIG_BT_EXT_ADV_MAX_ADV_SET elements.

Parameters

- adv – Advertising set.

Returns Index of the advertising set object. The range of the returned value is 0..CONFIG_BT_EXT_ADV_MAX_ADV_SET-1

int bt_le_ext_adv_get_info(const struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_info* *info)

Get advertising set info.

Parameters

- adv – Advertising set object
- info – Advertising set info object

Returns Zero on success or (negative) error code on failure.

int bt_le_per_adv_set_param(struct bt_le_ext_adv *adv, const struct *bt_le_per_adv_param* *param)

Set or update the periodic advertising parameters.

The periodic advertising parameters can only be set or updated on an extended advertisement set which is neither scannable, connectable nor anonymous.

Parameters

- adv – Advertising set object.
- param – Advertising parameters.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_set_data(const struct bt_le_ext_adv *adv, const struct *bt_data* *ad, size_t ad_len)

Set or update the periodic advertising data.

The periodic advertisement data can only be set or updated on an extended advertisement set which is neither scannable, connectable nor anonymous.

Parameters

- adv – Advertising set object.
- ad – Advertising data.
- ad_len – Advertising data length.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_start(struct bt_le_ext_adv *adv)

Starts periodic advertising.

Enabling the periodic advertising can be done independently of extended advertising, but both periodic advertising and extended advertising shall be enabled before any periodic advertising data is sent. The periodic advertising and extended advertising can be enabled in any order.

Once periodic advertising has been enabled, it will continue advertising until *bt_le_per_adv_stop()* has been called, or if the advertising set is deleted by *bt_le_ext_adv_delete()*. Calling *bt_le_ext_adv_stop()* will not stop the periodic advertising.

Parameters

- `adv` – Advertising set object.

Returns Zero on success or (negative) error code otherwise.

int `bt_le_per_adv_stop`(struct bt_le_ext_adv *adv)

Stops periodic advertising.

Disabling the periodic advertising can be done independently of extended advertising. Disabling periodic advertising will not disable extended advertising.

Parameters

- `adv` – Advertising set object.

Returns Zero on success or (negative) error code otherwise.

uint8_t `bt_le_per_adv_sync_get_index`(struct bt_le_per_adv_sync *per_adv_sync)

Get array index of an periodic advertising sync object.

This function is get the index of an array of periodic advertising sync objects. The array has CONFIG_BT_PER_ADV_SYNC_MAX elements.

Parameters

- `per_adv_sync` – The periodic advertising sync object.

Returns Index of the periodic advertising sync object. The range of the returned value is 0..CONFIG_BT_PER_ADV_SYNC_MAX-1

int `bt_le_per_adv_sync_get_info`(struct bt_le_per_adv_sync *per_adv_sync, struct *bt_le_per_adv_sync_info* *info)

Get periodic adv sync information.

Parameters

- `per_adv_sync` – Periodic advertising sync object.
- `info` – Periodic advertising sync info object

Returns Zero on success or (negative) error code on failure.

struct bt_le_per_adv_sync *`bt_le_per_adv_sync_lookup_addr`(const *bt_addr_le_t* *adv_addr, uint8_t sid)

Look up an existing periodic advertising sync object by advertiser address.

Parameters

- `adv_addr` – Advertiser address.
- `sid` – The advertising set ID.

Returns Periodic advertising sync object or NULL if not found.

int `bt_le_per_adv_sync_create`(const struct *bt_le_per_adv_sync_param* *param, struct bt_le_per_adv_sync **out_sync)

Create a periodic advertising sync object.

Create a periodic advertising sync object that can try to synchronize to periodic advertising reports from an advertiser. Scan shall either be disabled or extended scan shall be enabled.

Parameters

- `param` – **[in]** Periodic advertising sync parameters.
- `out_sync` – **[out]** Periodic advertising sync object on.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_sync_delete(struct bt_le_per_adv_sync *per_adv_sync)

Delete periodic advertising sync.

Delete the periodic advertising sync object. Can be called regardless of the state of the sync. If the syncing is currently syncing, the syncing is cancelled. If the sync has been established, it is terminated. The periodic advertising sync object will be invalidated afterwards.

If the state of the sync object is syncing, then a new periodic advertising sync object may not be created until the controller has finished canceling this object.

Parameters

- per_adv_sync – The periodic advertising sync object.

Returns Zero on success or (negative) error code otherwise.

void bt_le_per_adv_sync_cb_register(struct *bt_le_per_adv_sync_cb* *cb)

Register periodic advertising sync callbacks.

Adds the callback structure to the list of callback structures for periodic adverising syncs.

This callback will be called for all periodic advertising sync activity, such as synced, terminated and when data is received.

Parameters

- cb – Callback struct. Must point to memory that remains valid.

int bt_le_per_adv_sync_recv_enable(struct bt_le_per_adv_sync *per_adv_sync)

Enables receiving periodic advertising reports for a sync.

If the sync is already receiving the reports, -EALREADY is returned.

Parameters

- per_adv_sync – The periodic advertising sync object.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_sync_recv_disable(struct bt_le_per_adv_sync *per_adv_sync)

Disables receiving periodic advertising reports for a sync.

If the sync report receiving is already disabled, -EALREADY is returned.

Parameters

- per_adv_sync – The periodic advertising sync object.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_sync_transfer(const struct bt_le_per_adv_sync *per_adv_sync, const struct bt_conn *conn, uint16_t service_data)

Transfer the periodic advertising sync information to a peer device.

This will allow another device to quickly synchronize to the same periodic advertising train that this device is currently synced to.

Parameters

- per_adv_sync – The periodic advertising sync to transfer.
- conn – The peer device that will receive the sync information.
- service_data – Application service data provided to the remote host.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_set_info_transfer(const struct bt_le_ext_adv *adv, const struct bt_conn *conn, uint16_t service_data)

Transfer the information about a periodic advertising set.

This will allow another device to quickly synchronize to periodic advertising set from this device.

Parameters

- adv – The periodic advertising set to transfer info of.
- conn – The peer device that will receive the information.
- service_data – Application service data provided to the remote host.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_sync_transfer_subscribe(const struct bt_conn *conn, const struct *bt_le_per_adv_sync_transfer_param* *param)

Subscribe to periodic advertising sync transfers (PASTs).

Sets the parameters and allow other devices to transfer periodic advertising syncs.

Parameters

- conn – The connection to set the parameters for. If NULL default parameters for all connections will be set. Parameters set for specific connection will always have precedence.
- param – The periodic advertising sync transfer parameters.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_sync_transfer_unsubscribe(const struct bt_conn *conn)

Unsubscribe from periodic advertising sync transfers (PASTs).

Remove the parameters that allow other devices to transfer periodic advertising syncs.

Parameters

- conn – The connection to remove the parameters for. If NULL default parameters for all connections will be removed. Unsubscribing for a specific device, will still allow other devices to transfer periodic advertising syncs.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_list_add(const *bt_addr_le_t* *addr, uint8_t sid)

Add a device to the periodic advertising list.

Add peer device LE address to the periodic advertising list. This will make it possibly to automatically create a periodic advertising sync to this device.

Parameters

- addr – Bluetooth LE identity address.
- sid – The advertising set ID. This value is obtained from the *bt_le_scan_recv_info* in the scan callback.

Returns Zero on success or (negative) error code otherwise.

int bt_le_per_adv_list_remove(const *bt_addr_le_t* *addr, uint8_t sid)

Remove a device from the periodic advertising list.

Removes peer device LE address from the periodic advertising list.

Parameters

- addr – Bluetooth LE identity address.

- `sid` – The advertising set ID. This value is obtained from the *bt_le_scan_recv_info* in the scan callback.

Returns Zero on success or (negative) error code otherwise.

int `bt_le_per_adv_list_clear`(void)

Clear the periodic advertising list.

Clears the entire periodic advertising list.

Returns Zero on success or (negative) error code otherwise.

int `bt_le_scan_start`(const struct *bt_le_scan_param* *param, *bt_le_scan_cb_t* cb)

Start (LE) scanning.

Start LE scanning with given parameters and provide results through the specified callback.

Note: The LE scanner by default does not use the Identity Address of the local device when `CONFIG_BT_PRIVACY` is disabled. This is to prevent the active scanner from disclosing the identity information when requesting additional information from advertisers. In order to enable directed advertiser reports then `CONFIG_BT_SCAN_WITH_IDENTITY` must be enabled.

Parameters

- `param` – Scan parameters.
- `cb` – Callback to notify scan results. May be NULL if callback registration through *bt_le_scan_cb_register* is preferred.

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int `bt_le_scan_stop`(void)

Stop (LE) scanning.

Stops ongoing LE scanning.

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

void `bt_le_scan_cb_register`(struct *bt_le_scan_cb* *cb)

Register scanner packet callbacks.

Adds the callback structure to the list of callback structures that monitors scanner activity.

This callback will be called for all scanner activity, regardless of what API was used to start the scanner.

Parameters

- `cb` – Callback struct. Must point to memory that remains valid.

void `bt_le_scan_cb_unregister`(struct *bt_le_scan_cb* *cb)

Unregister scanner packet callbacks.

Remove the callback structure from the list of scanner callbacks.

Parameters

- `cb` – Callback struct. Must point to memory that remains valid.

int `bt_le_filter_accept_list_add`(const *bt_addr_le_t* *addr)

Add device (LE) to filter accept list.

Add peer device LE address to the filter accept list.

Note: The filter accept list cannot be modified when an LE role is using the filter accept list, i.e advertiser or scanner using a filter accept list or automatic connecting to devices using filter accept list.

Parameters

- `addr` – Bluetooth LE identity address.

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

static inline int `bt_le_whitelist_add`(const *bt_addr_le_t* *addr)

int `bt_le_filter_accept_list_remove`(const *bt_addr_le_t* *addr)

Remove device (LE) from filter accept list.

Remove peer device LE address from the filter accept list.

Note: The filter accept list cannot be modified when an LE role is using the filter accept list, i.e advertiser or scanner using a filter accept list or automatic connecting to devices using filter accept list.

Parameters

- `addr` – Bluetooth LE identity address.

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

static inline int `bt_le_whitelist_rem`(const *bt_addr_le_t* *addr)

int `bt_le_filter_accept_list_clear`(void)

Clear filter accept list.

Clear all devices from the filter accept list.

Note: The filter accept list cannot be modified when an LE role is using the filter accept list, i.e advertiser or scanner using a filter accept list or automatic connecting to devices using filter accept list.

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

static inline int `bt_le_whitelist_clear`(void)

int `bt_le_set_chan_map`(uint8_t chan_map[5])

Set (LE) channel map.

Parameters

- `chan_map` – Channel map.

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

void bt_data_parse(struct *net_buf_simple* *ad, bool (*func)(struct *bt_data* *data, void *user_data), void *user_data)

Helper for parsing advertising (or EIR or OOB) data.

A helper for parsing the basic data types used for Extended Inquiry Response (EIR), Advertising Data (AD), and OOB data blocks. The most common scenario is to call this helper on the advertising data received in the callback that was given to *bt_le_scan_start()*.

Parameters

- ad – Advertising data as given to the bt_le_scan_cb_t callback.
- func – Callback function which will be called for each element that's found in the data. The callback should return true to continue parsing, or false to stop parsing.
- user_data – User data to be passed to the callback.

int bt_le_oob_get_local(uint8_t id, struct *bt_le_oob* *oob)

Get local LE Out of Band (OOB) information.

This function allows to get local information that are useful for Out of Band pairing or connection creation.

If privacy CONFIG_BT_PRIVACY is enabled this will result in generating new Resolvable Private Address (RPA) that is valid for CONFIG_BT_RPA_TIMEOUT seconds. This address will be used for advertising started by *bt_le_adv_start*, active scanning and connection creation.

Note: If privacy is enabled the RPA cannot be refreshed in the following cases:

- Creating a connection in progress, wait for the connected callback. In addition when extended advertising CONFIG_BT_EXT_ADV is not enabled or not supported by the controller:
- Advertiser is enabled using a Random Static Identity Address for a different local identity.
- The local identity conflicts with the local identity used by other roles.

Parameters

- id – **[in]** Local identity, in most cases BT_ID_DEFAULT.
- oob – **[out]** LE OOB information

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int bt_le_ext_adv_oob_get_local(struct bt_le_ext_adv *adv, struct *bt_le_oob* *oob)

Get local LE Out of Band (OOB) information.

This function allows to get local information that are useful for Out of Band pairing or connection creation.

If privacy CONFIG_BT_PRIVACY is enabled this will result in generating new Resolvable Private Address (RPA) that is valid for CONFIG_BT_RPA_TIMEOUT seconds. This address will be used by the advertising set.

Note: When generating OOB information for multiple advertising set all OOB information needs to be generated at the same time.

Note: If privacy is enabled the RPA cannot be refreshed in the following cases:

- Creating a connection in progress, wait for the connected callback.

Parameters

- `adv` – **[in]** The advertising set object
- `oob` – **[out]** LE OOB information

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int `bt_br_discovery_start`(const struct *bt_br_discovery_param* *param, struct *bt_br_discovery_result* *results, size_t count, *bt_br_discovery_cb_t* cb)

Start BR/EDR discovery.

Start BR/EDR discovery (inquiry) and provide results through the specified callback. When bt_br_discovery_cb_t is called it indicates that discovery has completed. If more inquiry results were received during session than fits in provided result storage, only ones with highest RSSI will be reported.

Parameters

- `param` – Discovery parameters.
- `results` – Storage for discovery results.
- `count` – Number of results in storage. Valid range: 1-255.
- `cb` – Callback to notify discovery results.

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error

int `bt_br_discovery_stop`(void)

Stop BR/EDR discovery.

Stops ongoing BR/EDR discovery. If discovery was stopped by this call results won't be reported

Returns Zero on success or error code otherwise, positive in case of protocol error or negative (POSIX) in case of stack internal error.

int `bt_br_oob_get_local`(struct *bt_br_oob* *oob)

Get BR/EDR local Out Of Band information.

This function allows to get local controller information that are useful for Out Of Band pairing or connection creation process.

Parameters

- `oob` – Out Of Band information

int `bt_br_set_discoverable`(bool enable)

Enable/disable set controller in discoverable state.

Allows make local controller to listen on INQUIRY SCAN channel and responds to devices making general inquiry. To enable this state it's mandatory to first be in connectable state.

Parameters

- `enable` – Value allowing/disallowing controller to become discoverable.

Returns Negative if fail set to requested state or requested state has been already set. Zero if done successfully.

int bt_br_set_connectable(bool enable)

Enable/disable set controller in connectable state.

Allows make local controller to be connectable. It means the controller start listen to devices requests on PAGE SCAN channel. If disabled also resets discoverability if was set.

Parameters

- enable – Value allowing/disallowing controller to be connectable.

Returns Negative if fail set to requested state or requested state has been already set. Zero if done successfully.

int bt_unpair(uint8_t id, const *bt_addr_le_t* *addr)

Clear pairing information.

Parameters

- id – Local identity (mostly just BT_ID_DEFAULT).
- addr – Remote address, NULL or BT_ADDR_LE_ANY to clear all remote devices.

Returns 0 on success or negative error value on failure.

void bt_foreach_bond(uint8_t id, void (*func)(const struct *bt_bond_info* *info, void *user_data), void *user_data)

Iterate through all existing bonds.

Parameters

- id – Local identity (mostly just BT_ID_DEFAULT).
- func – Function to call for each bond.
- user_data – Data to pass to the callback function.

struct bt_le_ext_adv_sent_info

#include <bluetooth.h>

Public Members

uint8_t num_sent

The number of advertising events completed.

struct bt_le_ext_adv_connected_info

#include <bluetooth.h>

Public Members

struct bt_conn *conn

Connection object of the new connection

struct bt_le_ext_adv_scanned_info

#include <bluetooth.h>

Public Members

bt_addr_le_t *addr
Active scanner LE address and type

struct bt_le_ext_adv_cb
#include <bluetooth.h>

Public Members

void (*sent)(struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_sent_info* *info)
The advertising set has finished sending adv data.

This callback notifies the application that the advertising set has finished sending advertising data. The advertising set can either have been stopped by a timeout or because the specified number of advertising events has been reached.
Param adv The advertising set object.
Param info Information about the sent event.

void (*connected)(struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_connected_info* *info)
The advertising set has accepted a new connection.

This callback notifies the application that the advertising set has accepted a new connection.
Param adv The advertising set object.
Param info Information about the connected event.

void (*scanned)(struct bt_le_ext_adv *adv, struct *bt_le_ext_adv_scanned_info* *info)
The advertising set has sent scan response data.

This callback notifies the application that the advertising set has has received a Scan Request packet, and has sent a Scan Response packet.
Param adv The advertising set object.
Param addr Information about the scanned event.

struct bt_data
#include <bluetooth.h> Bluetooth data.

Description of different data types that can be encoded into advertising data. Used to form arrays that are passed to the *bt_le_adv_start()* function.

struct bt_le_adv_param
#include <bluetooth.h> LE Advertising Parameters.

Public Members

uint8_t id
Local identity.

Note: When extended advertising CONFIG_BT_EXT_ADV is not enabled or not supported by the controller it is not possible to scan and advertise simultaneously using two different random addresses.

---

uint8_t `sid`

Advertising Set Identifier, valid range 0x00 - 0x0f.

Note: Requires *BT_LE_ADV_OPT_EXT_ADV*

uint8_t `secondary_max_skip`

Secondary channel maximum skip count.

Maximum advertising events the advertiser can skip before it must send advertising data on the secondary advertising channel.

Note: Requires *BT_LE_ADV_OPT_EXT_ADV*

uint32_t `options`

Bit-field of advertising options

uint32_t `interval_min`

Minimum Advertising Interval (N * 0.625 milliseconds) Minimum Advertising Interval shall be less than or equal to the Maximum Advertising Interval. The Minimum Advertising Interval and Maximum Advertising Interval should not be the same value (as stated in Bluetooth Core Spec 5.2, section 7.8.5) Range: 0x0020 to 0x4000

uint32_t `interval_max`

Maximum Advertising Interval (N * 0.625 milliseconds) Minimum Advertising Interval shall be less than or equal to the Maximum Advertising Interval. The Minimum Advertising Interval and Maximum Advertising Interval should not be the same value (as stated in Bluetooth Core Spec 5.2, section 7.8.5) Range: 0x0020 to 0x4000

const *bt_addr_le_t* *`peer`

Directed advertising to peer.

When this parameter is set the advertiser will send directed advertising to the remote device.

The advertising type will either be high duty cycle, or low duty cycle if the BT_LE_ADV_OPT_DIR_MODE_LOW_DUTY option is enabled. When using *BT_LE_ADV_OPT_EXT_ADV* then only low duty cycle is allowed.

In case of connectable high duty cycle if the connection could not be established within the timeout the connected() callback will be called with the status set to BT_HCI_ERR_ADV_TIMEOUT.

struct `bt_le_per_adv_param`

#include <bluetooth.h>

Public Members

uint16_t `interval_min`

Minimum Periodic Advertising Interval (N * 1.25 ms)

Shall be greater or equal to BT_GAP_PER_ADV_MIN_INTERVAL and less or equal to interval_max.

uint16_t `interval_max`

Maximum Periodic Advertising Interval (N * 1.25 ms)

Shall be less or equal to BT_GAP_PER_ADV_MAX_INTERVAL and greater or equal to interval_min.

uint32_t `options`

Bit-field of periodic advertising options

struct `bt_le_ext_adv_start_param`

#include <bluetooth.h>

Public Members

uint16_t `timeout`

Advertiser timeout (N * 10 ms).

Application will be notified by the advertiser sent callback. Set to zero for no timeout.

When using high duty cycle directed connectable advertising then this parameters must be set to a non-zero value less than or equal to the maximum of *BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT*.

If privacy `CONFIG_BT_PRIVACY` is enabled then the timeout must be less than `CONFIG_BT_RPA_TIMEOUT` .

uint8_t `num_events`

Number of advertising events.

Application will be notified by the advertiser sent callback. Set to zero for no limit.

struct `bt_le_ext_adv_info`

#include <bluetooth.h> Advertising set info structure.

Public Members

int8_t `tx_power`

Currently selected Transmit Power (dBM).

struct `bt_le_per_adv_sync_synced_info`

#include <bluetooth.h>

Public Members

const *bt_addr_le_t* *`addr`

Advertiser LE address and type.

uint8_t `sid`

Advertiser SID

uint16_t `interval`
: Periodic advertising interval (N * 1.25 ms)

uint8_t `phy`
: Advertiser PHY

bool `recv_enabled`
: True if receiving periodic advertisements, false otherwise.

uint16_t `service_data`
: Service Data provided by the peer when sync is transferred.

  Will always be 0 when the sync is locally created.

struct bt_conn *`conn`
: Peer that transferred the periodic advertising sync.

  Will always be 0 when the sync is locally created.

struct `bt_le_per_adv_sync_term_info`
#include <bluetooth.h>

Public Members

const *bt_addr_le_t* *`addr`
: Advertiser LE address and type.

uint8_t `sid`
: Advertiser SID

uint8_t `reason`
: Cause of periodic advertising termination

struct `bt_le_per_adv_sync_recv_info`
#include <bluetooth.h>

Public Members

const *bt_addr_le_t* *`addr`
: Advertiser LE address and type.

uint8_t `sid`
: Advertiser SID

int8_t `tx_power`
: The TX power of the advertisement.

int8_t `rssi`

The RSSI of the advertisement excluding any CTE.

uint8_t `cte_type`

The Constant Tone Extension (CTE) of the advertisement (bt_df_cte_type)

struct `bt_le_per_adv_sync_state_info`

#include <bluetooth.h>

Public Members

bool `recv_enabled`

True if receiving periodic advertisements, false otherwise.

struct `bt_le_per_adv_sync_cb`

#include <bluetooth.h>

Public Members

void (*`synced`)(struct bt_le_per_adv_sync *sync, struct *bt_le_per_adv_sync_synced_info* *info)

The periodic advertising has been successfully synced.

This callback notifies the application that the periodic advertising set has been successfully synced, and will now start to receive periodic advertising reports.

Param sync The periodic advertising sync object.
Param info Information about the sync event.

void (*`term`)(struct bt_le_per_adv_sync *sync, const struct *bt_le_per_adv_sync_term_info* *info)

The periodic advertising sync has been terminated.

This callback notifies the application that the periodic advertising sync has been terminated, either by local request, remote request or because due to missing data, e.g. by being out of range or sync.

Param sync The periodic advertising sync object.

void (*`recv`)(struct bt_le_per_adv_sync *sync, const struct *bt_le_per_adv_sync_recv_info* *info, struct *net_buf_simple* *buf)

Periodic advertising data received.

This callback notifies the application of an periodic advertising report.

Param sync The advertising set object.
Param info Information about the periodic advertising event.
Param buf Buffer containing the periodic advertising data.

void (*`state_changed`)(struct bt_le_per_adv_sync *sync, const struct *bt_le_per_adv_sync_state_info* *info)

The periodic advertising sync state has changed.

This callback notifies the application about changes to the sync state. Initialize sync and termination is handled by their individual callbacks, and won't be notified here.

Param sync The periodic advertising sync object.
Param info Information about the state change.

void (*`biginfo`)(struct bt_le_per_adv_sync *sync, const struct bt_iso_biginfo *biginfo)

BIGInfo advertising report received.

This callback notifies the application of a BIGInfo advertising report. This is received if the advertiser is broadcasting isochronous streams in a BIG. See iso.h for more information.
Param sync The advertising set object.
Param biginfo The BIGInfo report.

void (*`cte_report_cb`)(struct bt_le_per_adv_sync *sync, struct bt_df_per_adv_sync_iq_samples_report const *info)

Callback for IQ samples report collected when sampling CTE received with periodic advertising PDU.
Param sync The periodic advertising sync object.
Param info Information about the sync event.

struct `bt_le_per_adv_sync_param`

#include <bluetooth.h>

Public Members

bt_addr_le_t `addr`

Periodic Advertiser Address.

Only valid if not using the periodic advertising list

uint8_t `sid`

Advertiser SID.

Only valid if not using the periodic advertising list

uint32_t `options`

Bit-field of periodic advertising sync options.

uint16_t `skip`

Maximum event skip.

Maximum number of periodic advertising events that can be skipped after a successful receive

uint16_t `timeout`

Synchronization timeout (N * 10 ms)

Synchronization timeout for the periodic advertising sync. Range 0x000A to 0x4000 (100 ms to 163840 ms)

struct `bt_le_per_adv_sync_info`

#include <bluetooth.h> Advertising set info structure.

Public Members

bt_addr_le_t addr
Periodic Advertiser Address

uint8_t sid
Advertiser SID

uint16_t interval
Periodic advertising interval (N * 1.25 ms)

uint8_t phy
Advertiser PHY

struct bt_le_per_adv_sync_transfer_param
#include <bluetooth.h>

Public Members

uint16_t skip
Maximum event skip.

The number of periodic advertising packets that can be skipped after a successful receive.

uint16_t timeout
Synchronization timeout (N * 10 ms)

Synchronization timeout for the periodic advertising sync. Range 0x000A to 0x4000 (100 ms to 163840 ms)

uint32_t options
Periodic Advertising Sync Transfer options

struct bt_le_scan_param
#include <bluetooth.h> LE scan parameters

Public Members

uint8_t type
Scan type (BT_LE_SCAN_TYPE_ACTIVE or BT_LE_SCAN_TYPE_PASSIVE)

uint32_t options
Bit-field of scanning options.

uint16_t interval
Scan interval (N * 0.625 ms)

uint16_t window
Scan window (N * 0.625 ms)

uint16_t `timeout`

Scan timeout (N * 10 ms)

Application will be notified by the scan timeout callback. Set zero to disable timeout.

uint16_t `interval_coded`

Scan interval LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan interval.

uint16_t `window_coded`

Scan window LE Coded PHY (N * 0.625 MS)

Set zero to use same as LE 1M PHY scan window.

struct `bt_le_scan_recv_info`

#include <bluetooth.h> LE advertisement packet information

Public Members

const *bt_addr_le_t* *`addr`

Advertiser LE address and type.

If advertiser is anonymous then this address will be *BT_ADDR_LE_ANY*.

uint8_t `sid`

Advertising Set Identifier.

int8_t `rssi`

Strength of advertiser signal.

int8_t `tx_power`

Transmit power of the advertiser.

uint8_t `adv_type`

Advertising packet type.

uint16_t `adv_props`

Advertising packet properties.

uint16_t `interval`

Periodic advertising interval.

If 0 there is no periodic advertising.

uint8_t `primary_phy`

Primary advertising channel PHY.

uint8_t `secondary_phy`

Secondary advertising channel PHY.

struct `bt_le_scan_cb`

#include <bluetooth.h> Listener context for (LE) scanning.

Public Members

void (*`recv`)(const struct *bt_le_scan_recv_info* *info, struct *net_buf_simple* *buf)

Advertisement packet received callback.

Param info Advertiser packet information.
Param buf Buffer containing advertiser data.

void (*`timeout`)(void)

The scanner has stopped scanning after scan timeout.

struct `bt_le_oob_sc_data`

#include <bluetooth.h> LE Secure Connections pairing Out of Band data.

Public Members

uint8_t `r`[16]

Random Number.

uint8_t `c`[16]

Confirm Value.

struct `bt_le_oob`

#include <bluetooth.h> LE Out of Band information.

Public Members

bt_addr_le_t `addr`

LE address. If privacy is enabled this is a Resolvable Private Address.

struct *bt_le_oob_sc_data* `le_sc_data`

LE Secure Connections pairing Out of Band data.

struct `bt_br_discovery_result`

#include <bluetooth.h> BR/EDR discovery result structure.

Public Members

bt_addr_t `addr`

Remote device address

int8_t `rssi`

RSSI from inquiry

uint8_t `cod`[3]
Class of Device

uint8_t `eir`[240]
Extended Inquiry Response

struct `bt_br_discovery_param`
#include <bluetooth.h> BR/EDR discovery parameters

Public Members

uint8_t `length`
Maximum length of the discovery in units of 1.28 seconds. Valid range is 0x01 - 0x30.

bool `limited`
True if limited discovery procedure is to be used.

struct `bt_br_oob`
#include <bluetooth.h>

Public Members

bt_addr_t `addr`
BR/EDR address.

struct `bt_bond_info`
#include <bluetooth.h> Information about a bond with a remote device.

Public Members

bt_addr_le_t `addr`
Address of the remote device.

group `bt_addr`
Bluetooth device address definitions and utilities.

Defines

`BT_ADDR_LE_PUBLIC`

`BT_ADDR_LE_RANDOM`

`BT_ADDR_LE_PUBLIC_ID`

BT_ADDR_LE_RANDOM_ID

BT_ADDR_ANY
: Bluetooth device “any” address, not a valid address

BT_ADDR_NONE
: Bluetooth device “none” address, not a valid address

BT_ADDR_LE_ANY
: Bluetooth LE device “any” address, not a valid address

BT_ADDR_LE_NONE
: Bluetooth LE device “none” address, not a valid address

BT_ADDR_IS_RPA(a)
: Check if a Bluetooth LE random address is resolvable private address.

BT_ADDR_IS_NRPA(a)
: Check if a Bluetooth LE random address is a non-resolvable private address.

BT_ADDR_IS_STATIC(a)
: Check if a Bluetooth LE random address is a static address.

BT_ADDR_SET_RPA(a)
: Set a Bluetooth LE random address as a resolvable private address.

BT_ADDR_SET_NRPA(a)
: Set a Bluetooth LE random address as a non-resolvable private address.

BT_ADDR_SET_STATIC(a)
: Set a Bluetooth LE random address as a static address.

BT_ADDR_STR_LEN
: Recommended length of user string buffer for Bluetooth address.

  The recommended length guarantee the output of address conversion will not lose valuable information about address being processed.

BT_ADDR_LE_STR_LEN
: Recommended length of user string buffer for Bluetooth LE address.

  The recommended length guarantee the output of address conversion will not lose valuable information about address being processed.

Functions

static inline int bt_addr_cmp(const *bt_addr_t* *a, const *bt_addr_t* *b)
: Compare Bluetooth device addresses.

  Parameters

  - a – First Bluetooth device address to compare
  - b – Second Bluetooth device address to compare

  Returns negative value if $a < b$, 0 if $a == b$, else positive

static inline int bt_addr_le_cmp(const *bt_addr_le_t* *a, const *bt_addr_le_t* *b)

Compare Bluetooth LE device addresses.

Parameters

- a – First Bluetooth LE device address to compare
- b – Second Bluetooth LE device address to compare

Returns negative value if $a < b$, 0 if $a == b$, else positive

static inline void bt_addr_copy(*bt_addr_t* *dst, const *bt_addr_t* *src)

Copy Bluetooth device address.

Parameters

- dst – Bluetooth device address destination buffer.
- src – Bluetooth device address source buffer.

static inline void bt_addr_le_copy(*bt_addr_le_t* *dst, const *bt_addr_le_t* *src)

Copy Bluetooth LE device address.

Parameters

- dst – Bluetooth LE device address destination buffer.
- src – Bluetooth LE device address source buffer.

int bt_addr_le_create_nrpa(*bt_addr_le_t* *addr)

Create a Bluetooth LE random non-resolvable private address.

int bt_addr_le_create_static(*bt_addr_le_t* *addr)

Create a Bluetooth LE random static address.

static inline bool bt_addr_le_is_rpa(const *bt_addr_le_t* *addr)

Check if a Bluetooth LE address is a random private resolvable address.

Parameters

- addr – Bluetooth LE device address.

Returns true if address is a random private resolvable address.

static inline bool bt_addr_le_is_identity(const *bt_addr_le_t* *addr)

Check if a Bluetooth LE address is valid identity address.

Valid Bluetooth LE identity addresses are either public address or random static address.

Parameters

- addr – Bluetooth LE device address.

Returns true if address is a valid identity address.

static inline int bt_addr_to_str(const *bt_addr_t* *addr, char *str, size_t len)

Converts binary Bluetooth address to string.

Parameters

- addr – Address of buffer containing binary Bluetooth address.
- str – Address of user buffer with enough room to store formatted string containing binary address.
- len – Length of data to be copied to user string buffer. Refer to BT_ADDR_STR_LEN about recommended value.

Returns Number of successfully formatted bytes from binary address.

static inline int bt_addr_le_to_str(const *bt_addr_le_t* *addr, char *str, size_t len)

Converts binary LE Bluetooth address to string.

Parameters

- addr – Address of buffer containing binary LE Bluetooth address.
- str – Address of user buffer with enough room to store formatted string containing binary LE address.
- len – Length of data to be copied to user string buffer. Refer to BT_ADDR_LE_STR_LEN about recommended value.

Returns Number of successfully formatted bytes from binary address.

int bt_addr_from_str(const char *str, *bt_addr_t* *addr)

Convert Bluetooth address from string to binary.

Parameters

- str – **[in]** The string representation of a Bluetooth address.
- addr – **[out]** Address of buffer to store the Bluetooth address

Returns Zero on success or (negative) error code otherwise.

int bt_addr_le_from_str(const char *str, const char *type, *bt_addr_le_t* *addr)

Convert LE Bluetooth address from string to binary.

Parameters

- str – **[in]** The string representation of an LE Bluetooth address.
- type – **[in]** The string representation of the LE Bluetooth address type.
- addr – **[out]** Address of buffer to store the LE Bluetooth address

Returns Zero on success or (negative) error code otherwise.

struct bt_addr_t

#include <*addr.h*> Bluetooth Device Address

struct bt_addr_le_t

#include <*addr.h*> Bluetooth LE Device Address

group bt_gap_defines

Bluetooth Generic Access Profile defines and Assigned Numbers.

Defines

BT_COMP_ID_LF

Company Identifiers (see Bluetooth Assigned Numbers)

BT_DATA_FLAGS

EIR/AD data type definitions

BT_DATA_UUID16_SOME

BT_DATA_UUID16_ALL

BT_DATA_UUID32_SOME

BT_DATA_UUID32_ALL

BT_DATA_UUID128_SOME

BT_DATA_UUID128_ALL

BT_DATA_NAME_SHORTENED

BT_DATA_NAME_COMPLETE

BT_DATA_TX_POWER

BT_DATA_SM_TK_VALUE

BT_DATA_SM_OOB_FLAGS

BT_DATA_SOLICIT16

BT_DATA_SOLICIT128

BT_DATA_SVC_DATA16

BT_DATA_GAP_APPEARANCE

BT_DATA_LE_BT_DEVICE_ADDRESS

BT_DATA_LE_ROLE

BT_DATA_SOLICIT32

BT_DATA_SVC_DATA32

BT_DATA_SVC_DATA128

BT_DATA_LE_SC_CONFIRM_VALUE

BT_DATA_LE_SC_RANDOM_VALUE

BT_DATA_URI

BT_DATA_CHANNEL_MAP_UPDATE_IND

BT_DATA_MESH_PROV

BT_DATA_MESH_MESSAGE

BT_DATA_MESH_BEACON

BT_DATA_BIG_INFO

BT_DATA_BROADCAST_CODE

BT_DATA_MANUFACTURER_DATA

BT_LE_AD_LIMITED

BT_LE_AD_GENERAL

BT_LE_AD_NO_BREDR

BT_GAP_SCAN_FAST_INTERVAL

BT_GAP_SCAN_FAST_WINDOW

BT_GAP_SCAN_SLOW_INTERVAL_1

BT_GAP_SCAN_SLOW_WINDOW_1

BT_GAP_SCAN_SLOW_INTERVAL_2

BT_GAP_SCAN_SLOW_WINDOW_2

BT_GAP_ADV_FAST_INT_MIN_1

BT_GAP_ADV_FAST_INT_MAX_1

BT_GAP_ADV_FAST_INT_MIN_2

BT_GAP_ADV_FAST_INT_MAX_2

BT_GAP_ADV_SLOW_INT_MIN

BT_GAP_ADV_SLOW_INT_MAX

BT_GAP_PER_ADV_FAST_INT_MIN_1

BT_GAP_PER_ADV_FAST_INT_MAX_1

BT_GAP_PER_ADV_FAST_INT_MIN_2

BT_GAP_PER_ADV_FAST_INT_MAX_2

BT_GAP_PER_ADV_SLOW_INT_MIN

BT_GAP_PER_ADV_SLOW_INT_MAX

BT_GAP_INIT_CONN_INT_MIN

BT_GAP_INIT_CONN_INT_MAX

BT_GAP_ADV_MAX_ADV_DATA_LEN

Maximum advertising data length.

BT_GAP_ADV_MAX_EXT_ADV_DATA_LEN

Maximum extended advertising data length.

Note: The maximum advertising data length that can be sent by an extended advertiser is defined by the controller.

BT_GAP_TX_POWER_INVALID

BT_GAP_RSSI_INVALID

BT_GAP_SID_INVALID

BT_GAP_NO_TIMEOUT

BT_GAP_ADV_HIGH_DUTY_CYCLE_MAX_TIMEOUT

BT_GAP_DATA_LEN_DEFAULT

BT_GAP_DATA_LEN_MAX

BT_GAP_DATA_TIME_DEFAULT

BT_GAP_DATA_TIME_MAX

BT_GAP_SID_MAX

BT_GAP_PER_ADV_MAX_SKIP

BT_GAP_PER_ADV_MIN_TIMEOUT

BT_GAP_PER_ADV_MAX_TIMEOUT

BT_GAP_PER_ADV_MIN_INTERVAL

Minimum Periodic Advertising Interval (N * 1.25 ms)

BT_GAP_PER_ADV_MAX_INTERVAL

Maximum Periodic Advertising Interval (N * 1.25 ms)

Enums

enum [anonymous]

LE PHY types

Values:

enumerator BT_GAP_LE_PHY_NONE = 0

Convenience macro for when no PHY is set.

enumerator BT_GAP_LE_PHY_1M = *BIT*(0)

LE 1M PHY

enumerator BT_GAP_LE_PHY_2M = *BIT*(1)

LE 2M PHY

enumerator BT_GAP_LE_PHY_CODED = *BIT*(2)

LE Coded PHY

enum [anonymous]

Advertising PDU types

Values:

enumerator BT_GAP_ADV_TYPE_ADV_IND = 0x00

Scannable and connectable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_DIRECT_IND = 0x01

Directed connectable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_SCAN_IND = 0x02

Non-connectable and scannable advertising.

enumerator BT_GAP_ADV_TYPE_ADV_NONCONN_IND = 0x03

Non-connectable and non-scannable advertising.

enumerator BT_GAP_ADV_TYPE_SCAN_RSP = 0x04

Additional advertising data requested by an active scanner.

enumerator BT_GAP_ADV_TYPE_EXT_ADV = 0x05

Extended advertising, see advertising properties.

enum [anonymous]

Advertising PDU properties

Values:

enumerator BT_GAP_ADV_PROP_CONNECTABLE = *BIT*(0)

Connectable advertising.

enumerator BT_GAP_ADV_PROP_SCANNABLE = *BIT*(1)

Scannable advertising.

enumerator BT_GAP_ADV_PROP_DIRECTED = *BIT*(2)

Directed advertising.

enumerator BT_GAP_ADV_PROP_SCAN_RESPONSE = *BIT*(3)

Additional advertising data requested by an active scanner.

enumerator BT_GAP_ADV_PROP_EXT_ADV = *BIT*(4)

Extended advertising.

enum [anonymous]

Constant Tone Extension (CTE) types

Values:

enumerator BT_GAP_CTE_AOA = 0x00

Angle of Arrival

enumerator BT_GAP_CTE_AOD_1US = 0x01

Angle of Departure with 1 us slots

enumerator BT_GAP_CTE_AOD_2US = 0x02

Angle of Departure with 2 us slots

enumerator BT_GAP_CTE_NONE = 0xFF

No extensions

enum [anonymous]

Peripheral sleep clock accuracy (SCA) in ppm (parts per million)

Values:

enumerator BT_GAP_SCA_UNKNOWN = 0

enumerator BT_GAP_SCA_251_500 = 0

enumerator BT_GAP_SCA_151_250 = 1

enumerator BT_GAP_SCA_101_150 = 2

enumerator BT_GAP_SCA_76_100 = 3

enumerator BT_GAP_SCA_51_75 = 4

enumerator BT_GAP_SCA_31_50 = 5

enumerator BT_GAP_SCA_21_30 = 6

enumerator BT_GAP_SCA_0_20 = 7

7.4.6 Generic Attribute Profile (GATT)

GATT layer manages the service database providing APIs for service registration and attribute declaration.

Services can be registered using *bt_gatt_service_register()* API which takes the *bt_gatt_service* struct that provides the list of attributes the service contains. The helper macro *BT_GATT_SERVICE()* can be used to declare a service.

Attributes can be declared using the *bt_gatt_attr* struct or using one of the helper macros:

BT_GATT_PRIMARY_SERVICE Declares a Primary Service.

BT_GATT_SECONDARY_SERVICE Declares a Secondary Service.

BT_GATT_INCLUDE_SERVICE Declares a Include Service.

BT_GATT_CHARACTERISTIC Declares a Characteristic.

BT_GATT_DESCRIPTOR Declares a Descriptor.

BT_GATT_ATTRIBUTE Declares an Attribute.

BT_GATT_CCC Declares a Client Characteristic Configuration.

BT_GATT_CEP Declares a Characteristic Extended Properties.

BT_GATT_CUD Declares a Characteristic User Format.

Each attribute contain a `uuid`, which describes their type, a `read` callback, a `write` callback and a set of permission. Both read and write callbacks can be set to NULL if the attribute permission don't allow their respective operations.

Note: Attribute `read` and `write` callbacks are called directly from RX Thread thus it is not recommended to block for long periods of time in them.

Attribute value changes can be notified using *bt_gatt_notify()* API, alternatively there is *bt_gatt_notify_cb()* where is is possible to pass a callback to be called when it is necessary to know the exact instant when the data has been transmitted over the air. Indications are supported by *bt_gatt_indicate()* API.

Client procedures can be enabled with the configuration option: `CONFIG_BT_GATT_CLIENT`

Discover procedures can be initiated with the use of *bt_gatt_discover()* API which takes the *bt_gatt_discover_params* struct which describes the type of discovery. The parameters also serves as a filter when setting the `uuid` field only attributes which matches will be discovered, in contrast setting it to NULL allows all attributes to be discovered.

Note: Caching discovered attributes is not supported.

Read procedures are supported by *bt_gatt_read()* API which takes the *bt_gatt_read_params* struct as parameters. In the parameters one or more attributes can be set, though setting multiple handles requires the option: CONFIG_BT_GATT_READ_MULTIPLE

Write procedures are supported by *bt_gatt_write()* API and takes *bt_gatt_write_params* struct as parameters. In case the write operation don't require a response *bt_gatt_write_without_response()* or *bt_gatt_write_without_response_cb()* APIs can be used, with the later working similarly to *bt_gatt_notify_cb()*.

Subscriptions to notification and indication can be initiated with use of *bt_gatt_subscribe()* API which takes *bt_gatt_subscribe_params* as parameters. Multiple subscriptions to the same attribute are supported so there could be multiple `notify` callback being triggered for the same attribute. Subscriptions can be removed with use of *bt_gatt_unsubscribe()* API.

Note: When subscriptions are removed `notify` callback is called with the data set to NULL.

API Reference

group `bt_gatt`

Generic Attribute Profile (GATT)

Defines

`BT_GATT_ERR(_att_err)`

Construct error return value for attribute read and write callbacks.

Parameters

- `_att_err` – ATT error code

Returns Appropriate error code for the attribute callbacks.

`BT_GATT_CHRC_BROADCAST`

Characteristic broadcast property.

Characteristic Properties Bit field values

If set, permits broadcasts of the Characteristic Value using Server Characteristic Configuration Descriptor.

`BT_GATT_CHRC_READ`

Characteristic read property.

If set, permits reads of the Characteristic Value.

`BT_GATT_CHRC_WRITE_WITHOUT_RESP`

Characteristic write without response property.

If set, permits write of the Characteristic Value without response.

`BT_GATT_CHRC_WRITE`

Characteristic write with response property.

If set, permits write of the Characteristic Value with response.

BT_GATT_CHRC_NOTIFY

Characteristic notify property.

If set, permits notifications of a Characteristic Value without acknowledgment.

BT_GATT_CHRC_INDICATE

Characteristic indicate property.

If set, permits indications of a Characteristic Value with acknowledgment.

BT_GATT_CHRC_AUTH

Characteristic Authenticated Signed Writes property.

If set, permits signed writes to the Characteristic Value.

BT_GATT_CHRC_EXT_PROP

Characteristic Extended Properties property.

If set, additional characteristic properties are defined in the Characteristic Extended Properties Descriptor.

BT_GATT_CEP_RELIABLE_WRITE

Characteristic Extended Properties Bit field values

BT_GATT_CEP_WRITABLE_AUX

BT_GATT_CCC_NOTIFY

Client Characteristic Configuration Notification.

Client Characteristic Configuration Values

If set, changes to Characteristic Value shall be notified.

BT_GATT_CCC_INDICATE

Client Characteristic Configuration Indication.

If set, changes to Characteristic Value shall be indicated.

BT_GATT_SCC_BROADCAST

Server Characteristic Configuration Broadcast.

Server Characteristic Configuration Values

If set, the characteristic value shall be broadcast in the advertising data when the server is advertising.

Enums

enum [anonymous]

GATT attribute permission bit field values

Values:

enumerator BT_GATT_PERM_NONE = 0

No operations supported, e.g. for notify-only

enumerator BT_GATT_PERM_READ = *BIT*(0)

Attribute read permission.

enumerator BT_GATT_PERM_WRITE = *BIT*(1)

Attribute write permission.

enumerator BT_GATT_PERM_READ_ENCRYPT = *BIT*(2)

Attribute read permission with encryption.

```
If set, requires encryption for read access.
```

enumerator BT_GATT_PERM_WRITE_ENCRYPT = *BIT*(3)

Attribute write permission with encryption.

```
If set, requires encryption for write access.
```

enumerator BT_GATT_PERM_READ_AUTHEN = *BIT*(4)

Attribute read permission with authentication.

```
If set, requires encryption using authenticated link-key for read
access.
```

enumerator BT_GATT_PERM_WRITE_AUTHEN = *BIT*(5)

Attribute write permission with authentication.

```
If set, requires encryption using authenticated link-key for write
access.
```

enumerator BT_GATT_PERM_PREPARE_WRITE = *BIT*(6)

Attribute prepare write permission.

```
If set, allows prepare writes with use of BT_GATT_WRITE_FLAG_PREPARE
passed to write callback.
```

enum [anonymous]

GATT attribute write flags

Values:

enumerator BT_GATT_WRITE_FLAG_PREPARE = *BIT*(0)

Attribute prepare write flag.

```
If set, write callback should only check if the device is
authorized but no data shall be written.
```

enumerator BT_GATT_WRITE_FLAG_CMD = *BIT*(1)

Attribute write command flag.

```
If set, indicates that write operation is a command (Write without
response) which doesn't generate any response.
```

struct `bt_gatt_attr`

#include <gatt.h> GATT Attribute structure.

Public Members

const struct *bt_uuid* *`uuid`

Attribute UUID

ssize_t (*`read`)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Attribute read callback.

The callback can also be used locally to read the contents of the attribute in which case no connection will be set.

Param conn The connection that is requesting to read
Param attr The attribute that's being read
Param buf Buffer to place the read result in
Param len Length of data to read
Param offset Offset to start reading from
Return Number fo bytes read, or in case of an error *BT_GATT_ERR()* with a specific ATT error code.

ssize_t (*`write`)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, const void *buf, uint16_t len, uint16_t offset, uint8_t flags)

Attribute write callback.

Param conn The connection that is requesting to write
Param attr The attribute that's being written
Param buf Buffer with the data to write
Param len Number of bytes in the buffer
Param offset Offset to start writing from
Param flags Flags (BT_GATT_WRITE_*)
Return Number of bytes written, or in case of an error *BT_GATT_ERR()* with a specific ATT error code.

void *`user_data`

Attribute user data

uint16_t `handle`

Attribute handle

uint8_t `perm`

Attribute permissions

struct `bt_gatt_service_static`

#include <gatt.h> GATT Service structure.

Public Members

const struct *bt_gatt_attr* *`attrs`

Service Attributes

size_t `attr_count`
Service Attribute count

struct `bt_gatt_service`
#include <gatt.h> GATT Service structure.

Public Members

struct *bt_gatt_attr* *`attrs`
Service Attributes

size_t `attr_count`
Service Attribute count

struct `bt_gatt_service_val`
#include <gatt.h> Service Attribute Value.

Public Members

const struct *bt_uuid* *`uuid`
Service UUID.

uint16_t `end_handle`
Service end handle.

struct `bt_gatt_include`
#include <gatt.h> Include Attribute Value.

Public Members

const struct *bt_uuid* *`uuid`
Service UUID.

uint16_t `start_handle`
Service start handle.

uint16_t `end_handle`
Service end handle.

struct `bt_gatt_cb`
#include <gatt.h> GATT callback structure.

Public Members

void (*att_mtu_updated)(struct bt_conn *conn, uint16_t tx, uint16_t rx)

The maximum ATT MTU on a connection has changed.

This callback notifies the application that the maximum TX or RX ATT MTU has increased.

Param conn Connection object.
Param tx Updated TX ATT MTU.
Param rx Updated RX ATT MTU.

struct bt_gatt_chrc

#include <gatt.h> Characteristic Attribute Value.

Public Members

const struct *bt_uuid* *uuid

Characteristic UUID.

uint16_t value_handle

Characteristic Value handle.

uint8_t properties

Characteristic properties.

struct bt_gatt_cep

#include <gatt.h> Characteristic Extended Properties Attribute Value.

Public Members

uint16_t properties

Characteristic Extended properties

struct bt_gatt_ccc

#include <gatt.h> Client Characteristic Configuration Attribute Value

Public Members

uint16_t flags

Client Characteristic Configuration flags

struct bt_gatt_scc

#include <gatt.h> Server Characterestic Configuration Attribute Value

Public Members

uint16_t flags

Server Characteristic Configuration flags

struct bt_gatt_cpf

#include <gatt.h> GATT Characteristic Presentation Format Attribute Value.

Public Members

uint8_t format

Format of the value of the characteristic

int8_t exponent

Exponent field to determine how the value of this characteristic is further formatted

uint16_t unit

Unit of the characteristic

uint8_t name_space

Name space of the description

uint16_t description

Description of the characteristic as defined in a higher layer profile

GATT Server

group bt_gatt_server

Defines

BT_GATT_SERVICE_DEFINE(_name, ...)

Statically define and register a service.

Helper macro to statically define and register a service.

Parameters

- _name – Service name.

BT_GATT_SERVICE_INSTANCE_DEFINE(_name, _instances, _instance_num, _attrs_def)

Statically define service structure array.

Helper macro to statically define service structure array. Each element of the array is linked to the service attribute array which is also defined in this scope using _attrs_def macro.

Parameters

- _name – Name of service structure array.
- _instances – Array of instances to pass as user context to the attribute callbacks.
- _instance_num – Number of elements in instance array.
- _attrs_def – Macro provided by the user that defines attribute array for the serivce. This macro should accept single parameter which is the instance context.

BT_GATT_SERVICE(_attrs)

Service Structure Declaration Macro.

Helper macro to declare a service structure.

Parameters

- _attrs – Service attributes.

BT_GATT_PRIMARY_SERVICE(_service)

Primary Service Declaration Macro.

Helper macro to declare a primary service attribute.

Parameters

- _service – Service attribute value.

BT_GATT_SECONDARY_SERVICE(_service)

Secondary Service Declaration Macro.

Helper macro to declare a secondary service attribute.

Parameters

- _service – Service attribute value.

BT_GATT_INCLUDE_SERVICE(_service_incl)

Include Service Declaration Macro.

Helper macro to declare database internal include service attribute.

Parameters

- _service_incl – the first service attribute of service to include

BT_GATT_CHRC_INIT(_uuid, _handle, _props)

BT_GATT_CHARACTERISTIC(_uuid, _props, _perm, _read, _write, _user_data)

Characteristic and Value Declaration Macro.

Helper macro to declare a characteristic attribute along with its attribute value.

Parameters

- _uuid – Characteristic attribute uuid.
- _props – Characteristic attribute properties.
- _perm – Characteristic Attribute access permissions.
- _read – Characteristic Attribute read callback.
- _write – Characteristic Attribute write callback.
- _user_data – Characteristic Attribute user data.

BT_GATT_CCC_MAX

BT_GATT_CCC_INITIALIZER(_changed, _write, _match)

Initialize Client Characteristic Configuration Declaration Macro.

Helper macro to initialize a Managed CCC attribute value.

Parameters

- _changed – Configuration changed callback.
- _write – Configuration write callback.
- _match – Configuration match callback.

BT_GATT_CCC_MANAGED(_ccc, _perm)

Managed Client Characteristic Configuration Declaration Macro.

Helper macro to declare a Managed CCC attribute.

Parameters

- _ccc – CCC attribute user data, shall point to a _bt_gatt_ccc.
- _perm – CCC access permissions.

BT_GATT_CCC(_changed, _perm)

Client Characteristic Configuration Declaration Macro.

Helper macro to declare a CCC attribute.

Parameters

- _changed – Configuration changed callback.
- _perm – CCC access permissions.

BT_GATT_CEP(_value)

Characteristic Extended Properties Declaration Macro.

Helper macro to declare a CEP attribute.

Parameters

- _value – Pointer to a struct *bt_gatt_cep*.

BT_GATT_CUD(_value, _perm)

Characteristic User Format Descriptor Declaration Macro.

Helper macro to declare a CUD attribute.

Parameters

- _value – User description NULL-terminated C string.
- _perm – Descriptor attribute access permissions.

BT_GATT_CPF(_value)

Characteristic Presentation Format Descriptor Declaration Macro.

Helper macro to declare a CPF attribute.

Parameters

- _value – Pointer to a struct *bt_gatt_cpf*.

BT_GATT_DESCRIPTOR(_uuid, _perm, _read, _write, _user_data)

Descriptor Declaration Macro.

Helper macro to declare a descriptor attribute.

Parameters

- _uuid – Descriptor attribute uuid.
- _perm – Descriptor attribute access permissions.
- _read – Descriptor attribute read callback.
- _write – Descriptor attribute write callback.
- _user_data – Descriptor attribute user data.

BT_GATT_ATTRIBUTE(_uuid, _perm, _read, _write, _user_data)

Attribute Declaration Macro.

Helper macro to declare an attribute.

Parameters

- _uuid – Attribute uuid.
- _perm – Attribute access permissions.
- _read – Attribute read callback.
- _write – Attribute write callback.
- _user_data – Attribute user data.

Typedefs

typedef uint8_t (*bt_gatt_attr_func_t)(const struct *bt_gatt_attr* *attr, uint16_t handle, void *user_data)

Attribute iterator callback.

Param attr Attribute found.

Param handle Attribute handle found.

Param user_data Data given.

Return BT_GATT_ITER_CONTINUE if should continue to the next attribute.

Return BT_GATT_ITER_STOP to stop.

typedef void (*bt_gatt_complete_func_t)(struct bt_conn *conn, void *user_data)

Notification complete result callback.

Param conn Connection object.

Param user_data Data passed in by the user.

typedef void (*bt_gatt_indicate_func_t)(struct bt_conn *conn, struct *bt_gatt_indicate_params* *params, uint8_t err)

Indication complete result callback.

Param conn Connection object.

Param params Indication params object.

Param err ATT error code

typedef void (*bt_gatt_indicate_params_destroy_t)(struct *bt_gatt_indicate_params* *params)

Enums

enum [anonymous]

Values:

enumerator BT_GATT_ITER_STOP = 0

enumerator BT_GATT_ITER_CONTINUE

Functions

void `bt_gatt_cb_register`(struct *bt_gatt_cb* *cb)

Register GATT callbacks.

Register callbacks to monitor the state of GATT.

Parameters

- `cb` – Callback struct.

int `bt_gatt_service_register`(struct *bt_gatt_service* *svc)

Register GATT service.

Register GATT service. Applications can make use of macros such as BT_GATT_PRIMARY_SERVICE, BT_GATT_CHARACTERISTIC, BT_GATT_DESCRIPTOR, etc.

When using `CONFIG_BT_SETTINGS` then all services that should have bond configuration loaded, i.e. CCC values, must be registered before calling *settings_load*.

When using `CONFIG_BT_GATT_CACHING` and `CONFIG_BT_SETTINGS` then all services that should be included in the GATT Database Hash calculation should be added before calling *settings_load*. All services registered after settings_load will trigger a new database hash calculation and a new hash stored.

Parameters

- `svc` – Service containing the available attributes

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_service_unregister`(struct *bt_gatt_service* *svc)

Unregister GATT service. *.

Parameters

- `svc` – Service to be unregistered.

Returns 0 in case of success or negative value in case of error.

void `bt_gatt_foreach_attr_type`(uint16_t start_handle, uint16_t end_handle, const struct *bt_uuid* *uuid, const void *attr_data, uint16_t num_matches, *bt_gatt_attr_func_t* func, void *user_data)

Attribute iterator by type.

Iterate attributes in the given range matching given UUID and/or data.

Parameters

- `start_handle` – Start handle.
- `end_handle` – End handle.
- `uuid` – UUID to match, passing NULL skips UUID matching.
- `attr_data` – Attribute data to match, passing NULL skips data matching.
- `num_matches` – Number matches, passing 0 makes it unlimited.
- `func` – Callback function.
- `user_data` – Data to pass to the callback.

static inline void `bt_gatt_foreach_attr`(uint16_t start_handle, uint16_t end_handle, *bt_gatt_attr_func_t* func, void *user_data)

Attribute iterator.

Iterate attributes in the given range.

Parameters

- `start_handle` – Start handle.
- `end_handle` – End handle.
- `func` – Callback function.
- `user_data` – Data to pass to the callback.

struct *bt_gatt_attr* *`bt_gatt_attr_next`(const struct *bt_gatt_attr* *attr)

Iterate to the next attribute.

Iterate to the next attribute following a given attribute.

Parameters

- `attr` – Current Attribute.

Returns The next attribute or NULL if it cannot be found.

struct *bt_gatt_attr* *`bt_gatt_find_by_uuid`(const struct *bt_gatt_attr* *attr, uint16_t attr_count, const struct *bt_uuid* *uuid)

Find Attribute by UUID.

Find the attribute with the matching UUID. To limit the search to a service set the attr to the service attributes and the attr_count to the service attribute count .

Parameters

- `attr` – Pointer to an attribute that serves as the starting point for the search of a match for the UUID. Passing NULL will search the entire range.
- `attr_count` – The number of attributes from the starting point to search for a match for the UUID. Set to 0 to search until the end.
- `uuid` – UUID to match.

uint16_t `bt_gatt_attr_get_handle`(const struct *bt_gatt_attr* *attr)

Get Attribute handle.

Parameters

- `attr` – Attribute object.

Returns Handle of the corresponding attribute or zero if the attribute could not be found.

uint16_t `bt_gatt_attr_value_handle`(const struct *bt_gatt_attr* *attr)

Get the handle of the characteristic value descriptor.

Note: The user_data of the attribute must of type *bt_gatt_chrc*.

Parameters

- `attr` – A Characteristic Attribute.

Returns the handle of the corresponding Characteristic Value. The value will be zero (the invalid handle) if `attr` was not a characteristic attribute.

ssize_t `bt_gatt_attr_read`(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t buf_len, uint16_t offset, const void *value, uint16_t value_len)

Generic Read Attribute value helper.

Read attribute value from local database storing the result into buffer.

Parameters

- `conn` – Connection object.
- `attr` – Attribute to read.
- `buf` – Buffer to store the value.
- `buf_len` – Buffer length.
- `offset` – Start offset.
- `value` – Attribute value.
- `value_len` – Length of the attribute value.

Returns number of bytes read in case of success or negative values in case of error.

ssize_t `bt_gatt_attr_read_service`(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Service Attribute helper.

Read service attribute value from local database storing the result into buffer after encoding it.

Note: Only use this with attributes which user_data is a *bt_uuid*.

Parameters

- `conn` – Connection object.
- `attr` – Attribute to read.
- `buf` – Buffer to store the value read.
- `len` – Buffer length.
- `offset` – Start offset.

Returns number of bytes read in case of success or negative values in case of error.

ssize_t `bt_gatt_attr_read_included`(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Include Attribute helper.

Read include service attribute value from local database storing the result into buffer after encoding it.

Note: Only use this with attributes which user_data is a *bt_gatt_include*.

Parameters

- `conn` – Connection object.
- `attr` – Attribute to read.
- `buf` – Buffer to store the value read.
- `len` – Buffer length.
- `offset` – Start offset.

Returns number of bytes read in case of success or negative values in case of error.

ssize_t bt_gatt_attr_read_chrc(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic Attribute helper.

Read characteristic attribute value from local database storing the result into buffer after encoding it.

Note: Only use this with attributes which user_data is a *bt_gatt_chrc*.

Parameters

- conn – Connection object.
- attr – Attribute to read.
- buf – Buffer to store the value read.
- len – Buffer length.
- offset – Start offset.

Returns number of bytes read in case of success or negative values in case of error.

ssize_t bt_gatt_attr_read_ccc(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Client Characteristic Configuration Attribute helper.

Read CCC attribute value from local database storing the result into buffer after encoding it.

Note: Only use this with attributes which user_data is a _bt_gatt_ccc.

Parameters

- conn – Connection object.
- attr – Attribute to read.
- buf – Buffer to store the value read.
- len – Buffer length.
- offset – Start offset.

Returns number of bytes read in case of success or negative values in case of error.

ssize_t bt_gatt_attr_write_ccc(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, const void *buf, uint16_t len, uint16_t offset, uint8_t flags)

Write Client Characteristic Configuration Attribute helper.

Write value in the buffer into CCC attribute.

Note: Only use this with attributes which user_data is a _bt_gatt_ccc.

Parameters

- conn – Connection object.
- attr – Attribute to read.
- buf – Buffer to store the value read.
- len – Buffer length.

- `offset` – Start offset.
- `flags` – Write flags.

Returns number of bytes written in case of success or negative values in case of error.

ssize_t `bt_gatt_attr_read_cep`(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic Extended Properties Attribute helper.

Read CEP attribute value from local database storing the result into buffer after encoding it.

Note: Only use this with attributes which user_data is a *bt_gatt_cep*.

Parameters

- `conn` – Connection object
- `attr` – Attribute to read
- `buf` – Buffer to store the value read
- `len` – Buffer length
- `offset` – Start offset

Returns number of bytes read in case of success or negative values in case of error.

ssize_t `bt_gatt_attr_read_cud`(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic User Description Descriptor Attribute helper.

Read CUD attribute value from local database storing the result into buffer after encoding it.

Note: Only use this with attributes which user_data is a NULL-terminated C string.

Parameters

- `conn` – Connection object
- `attr` – Attribute to read
- `buf` – Buffer to store the value read
- `len` – Buffer length
- `offset` – Start offset

Returns number of bytes read in case of success or negative values in case of error.

ssize_t `bt_gatt_attr_read_cpf`(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, void *buf, uint16_t len, uint16_t offset)

Read Characteristic Presentation format Descriptor Attribute helper.

Read CPF attribute value from local database storing the result into buffer after encoding it.

Note: Only use this with attributes which user_data is a bt_gatt_pf.

Parameters

- `conn` – Connection object

- `attr` – Attribute to read
- `buf` – Buffer to store the value read
- `len` – Buffer length
- `offset` – Start offset

Returns number of bytes read in case of success or negative values in case of error.

int `bt_gatt_notify_cb`(struct bt_conn *conn, struct *bt_gatt_notify_params* *params)

Notify attribute value change.

This function works in the same way as *bt_gatt_notify*. With the addition that after sending the notification the callback function will be called.

The callback is run from System Workqueue context. When called from the System Workqueue context this API will not wait for resources for the callback but instead return an error. The number of pending callbacks can be increased with the `CONFIG_BT_CONN_TX_MAX` option.

Alternatively it is possible to notify by UUID by setting it on the parameters, when using this method the attribute if provided is used as the start range when looking up for possible matches.

Parameters

- `conn` – Connection object.
- `params` – Notification parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_notify_multiple`(struct bt_conn *conn, uint16_t num_params, struct *bt_gatt_notify_params* *params)

Notify multiple attribute value change.

This function works in the same way as *bt_gatt_notify_cb*.

Parameters

- `conn` – Connection object.
- `num_params` – Number of notification parameters.
- `params` – Array of notification parameters.

Returns 0 in case of success or negative value in case of error.

static inline int `bt_gatt_notify`(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, const void *data, uint16_t len)

Notify attribute value change.

Send notification of attribute value change, if connection is NULL notify all peer that have notification enabled via CCC otherwise do a direct notification only the given connection.

The attribute object on the parameters can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC.

Parameters

- `conn` – Connection object.
- `attr` – Characteristic or Characteristic Value attribute.
- `data` – Pointer to Attribute data.
- `len` – Attribute value length.

Returns 0 in case of success or negative value in case of error.

static inline int bt_gatt_notify_uuid(struct bt_conn *conn, const struct *bt_uuid* *uuid, const struct *bt_gatt_attr* *attr, const void *data, uint16_t len)

Notify attribute value change by UUID.

Send notification of attribute value change, if connection is NULL notify all peer that have notification enabled via CCC otherwise do a direct notification only on the given connection.

The attribute object is the starting point for the search of the UUID.

Parameters

- `conn` – Connection object.
- `uuid` – The UUID. If the server contains multiple services with the same UUID, then the first occurrence, starting from the attr given, is used.
- `attr` – Pointer to an attribute that serves as the starting point for the search of a match for the UUID.
- `data` – Pointer to Attribute data.
- `len` – Attribute value length.

Returns 0 in case of success or negative value in case of error.

int bt_gatt_indicate(struct bt_conn *conn, struct *bt_gatt_indicate_params* *params)

Indicate attribute value change.

Send an indication of attribute value change. if connection is NULL indicate all peer that have notification enabled via CCC otherwise do a direct indication only the given connection.

The attribute object on the parameters can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC.

Alternatively it is possible to indicate by UUID by setting it on the parameters, when using this method the attribute if provided is used as the start range when looking up for possible matches.

Note: This procedure is asynchronous therefore the parameters need to remains valid while it is active. The procedure is active until the destroy callback is run.

Parameters

- `conn` – Connection object.
- `params` – Indicate parameters.

Returns 0 in case of success or negative value in case of error.

bool bt_gatt_is_subscribed(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, uint16_t ccc_value)

Check if connection have subscribed to attribute.

Check if connection has subscribed to attribute value change.

The attribute object can be the so called Characteristic Declaration, which is usually declared with BT_GATT_CHARACTERISTIC followed by BT_GATT_CCC, or the Characteristic Value Declaration which is automatically created after the Characteristic Declaration when using BT_GATT_CHARACTERISTIC, or the Client Characteristic Configuration Descriptor (CCCD) which is created by BT_GATT_CCC.

Parameters

- `conn` – Connection object.

- `attr` – Attribute object.
- `ccc_value` – The subscription type, either notifications or indications.

Returns true if the attribute object has been subscribed.

uint16_t `bt_gatt_get_mtu`(struct bt_conn *conn)

Get ATT MTU for a connection.

Get negotiated ATT connection MTU, note that this does not equal the largest amount of attribute data that can be transferred within a single packet.

Parameters

- `conn` – Connection object.

Returns MTU in bytes

struct `bt_gatt_ccc_cfg`

#include <gatt.h> GATT CCC configuration entry.

Public Members

uint8_t `id`

Local identity, BT_ID_DEFAULT in most cases.

bt_addr_le_t `peer`

Remote peer address.

uint16_t `value`

Configuration value.

struct `bt_gatt_notify_params`

#include <gatt.h>

Public Members

const struct *bt_uuid* *`uuid`

Notification Attribute UUID type.

Optional, use to search for an attribute with matching UUID when the attribute object pointer is not known.

const struct *bt_gatt_attr* *`attr`

Notification Attribute object.

Optional if uuid is provided, in this case it will be used as start range to search for the attribute with the given UUID.

const void *`data`

Notification Value data

uint16_t `len`

Notification Value length

bt_gatt_complete_func_t `func`

Notification Value callback

void *`user_data`

Notification Value callback user data

struct `bt_gatt_indicate_params`

#include <gatt.h> GATT Indicate Value parameters.

Public Members

const struct *bt_uuid* *`uuid`

Indicate Attribute UUID type.

Optional, use to search for an attribute with matching UUID when the attribute object pointer is not known.

const struct *bt_gatt_attr* *`attr`

Indicate Attribute object.

Optional if uuid is provided, in this case it will be used as start range to search for the attribute with the given UUID.

bt_gatt_indicate_func_t `func`

Indicate Value callback

bt_gatt_indicate_params_destroy_t `destroy`

Indicate operation complete callback

const void *`data`

Indicate Value data

uint16_t `len`

Indicate Value length

GATT Client

group `bt_gatt_client`

Typedefs

typedef uint8_t (*`bt_gatt_discover_func_t`)(struct bt_conn *conn, const struct *bt_gatt_attr* *attr, struct *bt_gatt_discover_params* *params)

Discover attribute callback function.

If discovery procedure has completed this callback will be called with attr set to NULL. This will not happen if procedure was stopped by returning BT_GATT_ITER_STOP.

The attribute object as well as its UUID and value objects are temporary and must be copied to in order to cache its information. Only the following fields of the attribute contains valid information:

- uuid UUID representing the type of attribute.
- handle Handle in the remote database.
- user_data The value of the attribute. Will be NULL when discovering descriptors

To be able to read the value of the discovered attribute the user_data must be cast to an appropriate type.

- *bt_gatt_service_val* when UUID is *BT_UUID_GATT_PRIMARY* or *BT_UUID_GATT_SECONDARY*.
- *bt_gatt_include* when UUID is *BT_UUID_GATT_INCLUDE*.
- *bt_gatt_chrc* when UUID is *BT_UUID_GATT_CHRC*.

Param conn Connection object.

Param attr Attribute found, or NULL if not found.

Param params Discovery parameters given.

Return BT_GATT_ITER_CONTINUE to continue discovery procedure.

Return BT_GATT_ITER_STOP to stop discovery procedure.

typedef uint8_t (*`bt_gatt_read_func_t`)(struct bt_conn *conn, uint8_t err, struct *bt_gatt_read_params* *params, const void *data, uint16_t length)

Read callback function.

Param conn Connection object.

Param err ATT error code.

Param params Read parameters used.

Param data Attribute value data. NULL means read has completed.

Param length Attribute value length.

Return BT_GATT_ITER_CONTINUE if should continue to the next attribute.

Return BT_GATT_ITER_STOP to stop.

typedef void (*`bt_gatt_write_func_t`)(struct bt_conn *conn, uint8_t err, struct *bt_gatt_write_params* *params)

Write callback function.

Param conn Connection object.

Param err ATT error code.

Param params Write parameters used.

typedef uint8_t (*`bt_gatt_notify_func_t`)(struct bt_conn *conn, struct *bt_gatt_subscribe_params* *params, const void *data, uint16_t length)

Notification callback function.

In the case of an empty notification, the `data` pointer will be non-NULL while the `length` will be 0, which is due to the special case where a `data` NULL pointer means unsubscribed.

Param conn Connection object. May be NULL, indicating that the peer is being unpaired

Param params Subscription parameters.

Param data Attribute value data. If NULL then subscription was removed.

Param length Attribute value length.

Return BT_GATT_ITER_CONTINUE to continue receiving value notifications. BT_GATT_ITER_STOP to unsubscribe from value notifications.

Enums

enum [anonymous]

GATT Discover types

Values:

enumerator BT_GATT_DISCOVER_PRIMARY

Discover Primary Services.

enumerator BT_GATT_DISCOVER_SECONDARY

Discover Secondary Services.

enumerator BT_GATT_DISCOVER_INCLUDE

Discover Included Services.

enumerator BT_GATT_DISCOVER_CHARACTERISTIC

Discover Characteristic Values.

```
Discover Characteristic Value and its properties.
```

enumerator BT_GATT_DISCOVER_DESCRIPTOR

Discover Descriptors.

```
Discover Attributes which are not services or characteristics.

@note The use of this type of discover is not recommended for
      discovering in ranges across multiple services/characteristics
      as it may incur in extra round trips.
```

enumerator BT_GATT_DISCOVER_ATTRIBUTE

Discover Attributes.

```
Discover Attributes of any type.

@note The use of this type of discover is not recommended for
      discovering in ranges across multiple services/characteristics
      as it may incur in more round trips.
```

enumerator BT_GATT_DISCOVER_STD_CHAR_DESC

Discover standard characteristic descriptor values.

```
Discover standard characterestic descriptor values and their
properties.
Supported descriptors:
 - Characteristic Extended Properties
 - Client Characteristic Configuration
 - Server Characteristic Configuration
 - Characteristic Presentation Format
```

enum [anonymous]

Subscription flags

Values:

enumerator BT_GATT_SUBSCRIBE_FLAG_VOLATILE

Persistence flag.

```
If set, indicates that the subscription is not saved
on the GATT server side. Therefore, upon disconnection,
the subscription will be automatically removed
from the client's subscriptions list and
when the client reconnects, it will have to
issue a new subscription.
```

enumerator BT_GATT_SUBSCRIBE_FLAG_NO_RESUB

No resubscribe flag.

```
By default when BT_GATT_SUBSCRIBE_FLAG_VOLATILE is unset, the
subscription will be automatically renewed when the client
reconnects, as a workaround for GATT servers that do not persist
subscriptions.

This flag will disable the automatic resubscription. It is useful
if the application layer knows that the GATT server remembers
subscriptions from previous connections and wants to avoid renewing
the subscriptions.
```

enumerator BT_GATT_SUBSCRIBE_FLAG_WRITE_PENDING

Write pending flag.

```
If set, indicates write operation is pending waiting remote end to
respond.
```

enumerator BT_GATT_SUBSCRIBE_NUM_FLAGS

Functions

int bt_gatt_exchange_mtu(struct bt_conn *conn, struct *bt_gatt_exchange_params* *params)

Exchange MTU.

This client procedure can be used to set the MTU to the maximum possible size the buffers can hold.

Note: Shall only be used once per connection.

Parameters

- `conn` – Connection object.
- `params` – Exchange MTU parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_discover`(struct bt_conn *conn, struct *bt_gatt_discover_params* *params)

GATT Discover function.

This procedure is used by a client to discover attributes on a server.

Primary Service Discovery: Procedure allows to discover specific Primary Service based on UUID. Include Service Discovery: Procedure allows to discover all Include Services within specified range. Characteristic Discovery: Procedure allows to discover all characteristics within specified handle range as well as discover characteristics with specified UUID. Descriptors Discovery: Procedure allows to discover all characteristic descriptors within specified range.

For each attribute found the callback is called which can then decide whether to continue discovering or stop.

Note: This procedure is asynchronous therefore the parameters need to remains valid while it is active.

Parameters

- `conn` – Connection object.
- `params` – Discover parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_read`(struct bt_conn *conn, struct *bt_gatt_read_params* *params)

Read Attribute Value by handle.

This procedure read the attribute value and return it to the callback.

When reading attributes by UUID the callback can be called multiple times depending on how many instances of given the UUID exists with the start_handle being updated for each instance.

If an instance does contain a long value which cannot be read entirely the caller will need to read the remaining data separately using the handle and offset.

Note: This procedure is asynchronous therefore the parameters need to remains valid while it is active.

Parameters

- `conn` – Connection object.
- `params` – Read parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_write`(struct bt_conn *conn, struct *bt_gatt_write_params* *params)

Write Attribute Value by handle.

This procedure write the attribute value and return the result in the callback.

Note: This procedure is asynchronous therefore the parameters need to remains valid while it is active.

Parameters

- `conn` – Connection object.
- `params` – Write parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_write_without_response_cb`(struct bt_conn *conn, uint16_t handle, const void *data, uint16_t length, bool sign, *bt_gatt_complete_func_t* func, void *user_data)

Write Attribute Value by handle without response with callback.

This function works in the same way as *bt_gatt_write_without_response*. With the addition that after sending the write the callback function will be called.

The callback is run from System Workqueue context. When called from the System Workqueue context this API will not wait for resources for the callback but instead return an error. The number of pending callbacks can be increased with the `CONFIG_BT_CONN_TX_MAX` option.

Note: By using a callback it also disable the internal flow control which would prevent sending multiple commands without waiting for their transmissions to complete, so if that is required the caller shall not submit more data until the callback is called.

Parameters

- `conn` – Connection object.
- `handle` – Attribute handle.
- `data` – Data to be written.
- `length` – Data length.
- `sign` – Whether to sign data
- `func` – Transmission complete callback.
- `user_data` – User data to be passed back to callback.

Returns 0 in case of success or negative value in case of error.

static inline int `bt_gatt_write_without_response`(struct bt_conn *conn, uint16_t handle, const void *data, uint16_t length, bool sign)

Write Attribute Value by handle without response.

This procedure write the attribute value without requiring an acknowledgment that the write was successfully performed

Parameters

- `conn` – Connection object.
- `handle` – Attribute handle.
- `data` – Data to be written.
- `length` – Data length.
- `sign` – Whether to sign data

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_subscribe`(struct bt_conn *conn, struct *bt_gatt_subscribe_params* *params)

Subscribe Attribute Value Notification.

This procedure subscribe to value notification using the Client Characteristic Configuration handle. If notification received subscribe value callback is called to return notified value. One may then decide whether to unsubscribe directly from this callback. Notification callback with NULL data will not be called if subscription was removed by this method.

Note: Notifications are asynchronous therefore the parameters need to remain valid while subscribed.

Parameters

- `conn` – Connection object.
- `params` – Subscribe parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_resubscribe`(uint8_t id, const *bt_addr_le_t* *peer, struct *bt_gatt_subscribe_params* *params)

Resubscribe Attribute Value Notification subscription.

Resubscribe to Attribute Value Notification when already subscribed from a previous connection. The GATT server will remember subscription from previous connections when bonded, so resubscribing can be done without performing a new subscribe procedure after a power cycle.

Note: Notifications are asynchronous therefore the parameters need to remain valid while subscribed.

Parameters

- `id` – Local identity (in most cases BT_ID_DEFAULT).
- `peer` – Remote address.
- `params` – Subscribe parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_gatt_unsubscribe`(struct bt_conn *conn, struct *bt_gatt_subscribe_params* *params)

Unsubscribe Attribute Value Notification.

This procedure unsubscribe to value notification using the Client Characteristic Configuration handle. Notification callback with NULL data will be called if subscription was removed by this call, until then the parameters cannot be reused.

Parameters

- `conn` – Connection object.
- `params` – Subscribe parameters.

Returns 0 in case of success or negative value in case of error.

void `bt_gatt_cancel`(struct bt_conn *conn, void *params)

Cancel GATT pending request.

Parameters

- `conn` – Connection object.
- `params` – Requested params address.

struct `bt_gatt_exchange_params`

#include <gatt.h> GATT Exchange MTU parameters.

Public Members

void (*`func`)(struct bt_conn *conn, uint8_t err, struct *bt_gatt_exchange_params* *params)

Response callback

struct `bt_gatt_discover_params`

#include <gatt.h> GATT Discover Attributes parameters.

Public Members

const struct *bt_uuid* *`uuid`

Discover UUID type

bt_gatt_discover_func_t `func`

Discover attribute callback

uint16_t `attr_handle`

Include service attribute declaration handle

uint16_t `start_handle`

Included service start handle

Discover start handle

uint16_t `end_handle`

Included service end handle

Discover end handle

uint8_t `type`

Discover type

struct `bt_gatt_read_params`

#include <gatt.h> GATT Read parameters.

Public Members

bt_gatt_read_func_t `func`

Read attribute callback.

size_t `handle_count`

If equals to 1 single.handle and single.offset are used. If greater than 1 multiple.handles are used. If equals to 0 by_uuid is used for Read Using Characteristic UUID.

uint16_t `handle`

Attribute handle.

uint16_t `offset`

Attribute data offset.

uint16_t *`handles`

Attribute handles to read with Read Multiple Characteristic Values.

bool `variable`

If true use Read Multiple Variable Length Characteristic Values procedure. The values of the set of attributes may be of variable or unknown length. If false use Read Multiple Characteristic Values procedure. The values of the set of attributes must be of a known fixed length, with the exception of the last value that can have a variable length.

uint16_t `start_handle`

First requested handle number.

uint16_t `end_handle`

Last requested handle number.

const struct *bt_uuid* *`uuid`

2 or 16 octet UUID.

struct `bt_gatt_write_params`

#include <gatt.h> GATT Write parameters.

Public Members

bt_gatt_write_func_t `func`

Response callback

uint16_t `handle`

Attribute handle

uint16_t `offset`

Attribute data offset

const void *`data`

Data to be written

uint16_t `length`

Length of the data

struct bt_gatt_subscribe_params

#include <gatt.h> GATT Subscribe parameters.

Public Members

bt_gatt_notify_func_t notify

Notification value callback

bt_gatt_write_func_t write

Subscribe CCC write request response callback

uint16_t value_handle

Subscribe value handle

uint16_t ccc_handle

Subscribe CCC handle

uint16_t value

Subscribe value

bt_security_t min_security

Minimum required security for received notification. Notifications and indications received over a connection with a lower security level are silently discarded.

atomic_t flags[*ATOMIC_BITMAP_SIZE*(*BT_GATT_SUBSCRIBE_NUM_FLAGS*)]

Subscription flags

7.4.7 HCI Drivers

API Reference

group bt_hci_driver

HCI drivers.

Defines

IS_BT_QUIRK_NO_AUTO_DLE(bt_dev)

BT_HCI_EVT_FLAG_RECV_PRIO

BT_HCI_EVT_FLAG_RECV

Enums

enum [anonymous]

Values:

enumerator BT_QUIRK_NO_RESET = *BIT*(0)

enumerator BT_QUIRK_NO_AUTO_DLE = *BIT*(1)

enum bt_hci_driver_bus

Possible values for the 'bus' member of the *bt_hci_driver* struct

Values:

enumerator BT_HCI_DRIVER_BUS_VIRTUAL = 0

enumerator BT_HCI_DRIVER_BUS_USB = 1

enumerator BT_HCI_DRIVER_BUS_PCCARD = 2

enumerator BT_HCI_DRIVER_BUS_UART = 3

enumerator BT_HCI_DRIVER_BUS_RS232 = 4

enumerator BT_HCI_DRIVER_BUS_PCI = 5

enumerator BT_HCI_DRIVER_BUS_SDIO = 6

enumerator BT_HCI_DRIVER_BUS_SPI = 7

enumerator BT_HCI_DRIVER_BUS_I2C = 8

enumerator BT_HCI_DRIVER_BUS_IPM = 9

Functions

static inline uint8_t bt_hci_evt_get_flags(uint8_t evt)

Get HCI event flags.

Helper for the HCI driver to get HCI event flags that describes rules that. must be followed.

When CONFIG_BT_RECV_IS_RX_THREAD is enabled the flags BT_HCI_EVT_FLAG_RECV and BT_HCI_EVT_FLAG_RECV_PRIO indicates if the event should be given to bt_recv or bt_recv_prio.

Parameters

- evt – HCI event code.

Returns HCI event flags for the specified event.

int bt_recv(struct *net_buf* *buf)

Receive data from the controller/HCI driver.

This is the main function through which the HCI driver provides the host with data from the controller. The buffer needs to have its type set with the help of *bt_buf_set_type()* before calling this API.

When CONFIG_BT_RECV_IS_RX_THREAD is defined then this API should not be used for so-called high priority HCI events, which should instead be delivered to the host stack through *bt_recv_prio()*.

Parameters

- `buf` – Network buffer containing data from the controller.

Returns 0 on success or negative error number on failure.

int `bt_recv_prio`(struct *net_buf* *buf)

Receive high priority data from the controller/HCI driver.

This is the same as *bt_recv()*, except that it should be used for so-called high priority HCI events. There's a separate *bt_hci_evt_get_flags()* helper that can be used to identify which events have the BT_HCI_EVT_FLAG_RECV_PRIO flag set.

As with *bt_recv()*, the buffer needs to have its type set with the help of *bt_buf_set_type()* before calling this API. The only exception is so called high priority HCI events which should be delivered to the host stack through *bt_recv_prio()* instead.

Parameters

- `buf` – Network buffer containing data from the controller.

Returns 0 on success or negative error number on failure.

uint8_t `bt_read_static_addr`(struct bt_hci_vs_static_addr addrs[], uint8_t size)

Read static addresses from the controller.

Parameters

- `addrs` – Random static address and Identity Root (IR) array.
- `size` – Size of array.

Returns Number of addresses read.

int `bt_hci_driver_register`(const struct *bt_hci_driver* *drv)

Register a new HCI driver to the Bluetooth stack.

This needs to be called before any application code runs. The *bt_enable()* API will fail if there is no driver registered.

Parameters

- `drv` – A *bt_hci_driver* struct representing the driver.

Returns 0 on success or negative error number on failure.

int `bt_hci_transport_setup`(const struct *device* *dev)

Setup the HCI transport, which usually means to reset the Bluetooth IC.

Note: A weak version of this function is included in the H4 driver, so defining it is optional per board.

Parameters

- `dev` – The device structure for the bus connecting to the IC

Returns 0 on success, negative error value on failure

struct *net_buf* *bt_hci_evt_create(uint8_t evt, uint8_t len)

Allocate an HCI event buffer.

This function allocates a new buffer for an HCI event. It is given the avent code and the total length of the parameters. Upon successful return the buffer is ready to have the parameters encoded into it.

Parameters

- evt – Event OpCode.
- len – Length of event parameters.

Returns Newly allocated buffer.

struct *net_buf* *bt_hci_cmd_complete_create(uint16_t op, uint8_t plen)

Allocate an HCI Command Complete event buffer.

This function allocates a new buffer for HCI Command Complete event. It is given the OpCode (encoded e.g. using the BT_OP macro) and the total length of the parameters. Upon successful return the buffer is ready to have the parameters encoded into it.

Parameters

- op – Command OpCode.
- plen – Length of command parameters.

Returns Newly allocated buffer.

struct *net_buf* *bt_hci_cmd_status_create(uint16_t op, uint8_t status)

Allocate an HCI Command Status event buffer.

This function allocates a new buffer for HCI Command Status event. It is given the OpCode (encoded e.g. using the BT_OP macro) and the status code. Upon successful return the buffer is ready to have the parameters encoded into it.

Parameters

- op – Command OpCode.
- status – Status code.

Returns Newly allocated buffer.

struct bt_hci_driver

#include <hci_driver.h> Abstraction which represents the HCI transport to the controller.

This struct is used to represent the HCI transport to the Bluetooth controller.

Public Members

const char *name

Name of the driver

enum *bt_hci_driver_bus* bus

Bus of the transport (BT_HCI_DRIVER_BUS_*)

uint32_t quirks

Specific controller quirks. These are set by the HCI driver and acted upon by the host. They can either be statically set at buildtime, or set at runtime before the HCI driver's *open()* callback returns.

int (*open)(void)

Open the HCI transport.

Opens the HCI transport for operation. This function must not return until the transport is ready for operation, meaning it is safe to start calling the *send()* handler.

If the driver uses its own RX thread, i.e. CONFIG_BT_RECV_IS_RX_THREAD is set, then this function is expected to start that thread.

Return 0 on success or negative error number on failure.

int (*send)(struct *net_buf* *buf)

Send HCI buffer to controller.

Send an HCI command or ACL data to the controller. The exact type of the data can be checked with the help of *bt_buf_get_type()*.

Note: This function must only be called from a cooperative thread.

Param buf Buffer containing data to be sent to the controller.
Return 0 on success or negative error number on failure.

7.4.8 HCI RAW channel

Overview

HCI RAW channel API is intended to expose HCI interface to the remote entity. The local Bluetooth controller gets owned by the remote entity and host Bluetooth stack is not used. RAW API provides direct access to packets which are sent and received by the Bluetooth HCI driver.

API Reference

group hci_raw

HCI RAW channel.

Defines

BT_HCI_ERR_EXT_HANDLED

BT_HCI_RAW_CMD_EXT(_op, _min_len, _func)

Helper macro to define a command extension

Parameters

- _op – Opcode of the command.
- _min_len – Minimal length of the command.
- _func – Handler function to be called.

Enums

enum [anonymous]

Values:

enumerator BT_HCI_RAW_MODE_PASSTHROUGH = 0x00

Passthrough mode

```
While in this mode the buffers are passed as is between the stack
and the driver.
```

enumerator BT_HCI_RAW_MODE_H4 = 0x01

H:4 mode

```
While in this mode H:4 headers will added into the buffers
according to the buffer type when coming from the stack and will be
removed and used to set the buffer type.
```

Functions

int bt_send(struct *net_buf* *buf)

Send packet to the Bluetooth controller.

Send packet to the Bluetooth controller. Caller needs to implement netbuf pool.

Parameters

- buf – netbuf packet to be send

Returns Zero on success or (negative) error code otherwise.

int bt_hci_raw_set_mode(uint8_t mode)

Set Bluetooth RAW channel mode.

Set access mode of Bluetooth RAW channel.

Parameters

- mode – Access mode.

Returns Zero on success or (negative) error code otherwise.

uint8_t bt_hci_raw_get_mode(void)

Get Bluetooth RAW channel mode.

Get access mode of Bluetooth RAW channel.

Returns Access mode.

void bt_hci_raw_cmd_ext_register(struct *bt_hci_raw_cmd_ext* *cmds, size_t size)

Register Bluetooth RAW command extension table.

Register Bluetooth RAW channel command extension table, opcodes in this table are intercepted to sent to the handler function.

Parameters

- cmds – Pointer to the command extension table.
- size – Size of the command extension table.

int bt_enable_raw(struct k_fifo *rx_queue)

Enable Bluetooth RAW channel:

Enable Bluetooth RAW HCI channel.

Parameters

- `rx_queue` – netbuf queue where HCI packets received from the Bluetooth controller are to be queued. The queue is defined in the caller while the available buffers pools are handled in the stack.

Returns Zero on success or (negative) error code otherwise.

struct `bt_hci_raw_cmd_ext`

#include <hci_raw.h>

Public Members

uint16_t `op`

Opcode of the command

size_t `min_len`

Minimal length of the command

uint8_t (*`func`)(struct *net_buf* *buf)

Handler function.

Handler function to be called when a command is intercepted.

Param buf Buffer containing the command.

Return HCI Status code or BT_HCI_ERR_EXT_HANDLED if command has been handled already and a response has been sent as oppose to BT_HCI_ERR_SUCCESS which just indicates that the command can be sent to the controller to be processed.

7.4.9 Hands Free Profile (HFP)

API Reference

group `bt_hfp`

Hands Free Profile (HFP)

Defines

`HFP_HF_CMD_OK`

`HFP_HF_CMD_ERROR`

`HFP_HF_CMD_CME_ERROR`

`HFP_HF_CMD_UNKNOWN_ERROR`

Enums

enum `bt_hfp_hf_at_cmd`

Values:

enumerator `BT_HFP_HF_ATA`

enumerator `BT_HFP_HF_AT_CHUP`

Functions

int `bt_hfp_hf_register`(struct *bt_hfp_hf_cb* *cb)

Register HFP HF profile.

Register Handsfree profile callbacks to monitor the state and get the required HFP details to display.

Parameters

- `cb` – callback structure.

Returns 0 in case of success or negative value in case of error.

int `bt_hfp_hf_send_cmd`(struct bt_conn *conn, enum *bt_hfp_hf_at_cmd* cmd)

Handsfree client Send AT.

Send specific AT commands to handsfree client profile.

Parameters

- `conn` – Connection object.
- `cmd` – AT command to be sent.

Returns 0 in case of success or negative value in case of error.

struct `bt_hfp_hf_cmd_complete`

#include <hfp_hf.h> HFP HF Command completion field.

struct `bt_hfp_hf_cb`

#include <hfp_hf.h> HFP profile application callback.

Public Members

void (*`connected`)(struct bt_conn *conn)

HF connected callback to application

If this callback is provided it will be called whenever the connection completes.

Param conn Connection object.

void (*`disconnected`)(struct bt_conn *conn)

HF disconnected callback to application

If this callback is provided it will be called whenever the connection gets disconnected, including when a connection gets rejected or cancelled or any error in SLC establisment.

Param conn Connection object.

void (*service)(struct bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides service indicator value to the application

Param conn Connection object.
Param value service indicator value received from the AG.

void (*call)(struct bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides call indicator value to the application

Param conn Connection object.
Param value call indicator value received from the AG.

void (*call_setup)(struct bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides call setup indicator value to the application

Param conn Connection object.
Param value call setup indicator value received from the AG.

void (*call_held)(struct bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides call held indicator value to the application

Param conn Connection object.
Param value call held indicator value received from the AG.

void (*signal)(struct bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides signal indicator value to the application

Param conn Connection object.
Param value signal indicator value received from the AG.

void (*roam)(struct bt_conn *conn, uint32_t value)

HF indicator Callback

This callback provides roaming indicator value to the application

Param conn Connection object.
Param value roaming indicator value received from the AG.

void (*battery)(struct bt_conn *conn, uint32_t value)

HF indicator Callback

This callback battery service indicator value to the application

Param conn Connection object.
Param value battery indicator value received from the AG.

void (*ring_indication)(struct bt_conn *conn)

HF incoming call Ring indication callback to application

If this callback is provided it will be called whenever there is an incoming call.

Param conn Connection object.

void (*cmd_complete_cb)(struct bt_conn *conn, struct *bt_hfp_hf_cmd_complete* *cmd)

HF notify command completed callback to application

The command sent from the application is notified about its status
Param conn Connection object.
Param cmd structure contains status of the command including cme.

7.4.10 Logical Link Control and Adaptation Protocol (L2CAP)

L2CAP layer enables connection-oriented channels which can be enable with the configuration option: `CONFIG_BT_L2CAP_DYNAMIC_CHANNEL`. This channels support segmentation and reassembly transparently, they also support credit based flow control making it suitable for data streams.

Channels instances are represented by the *bt_l2cap_chan* struct which contains the callbacks in the *bt_l2cap_chan_ops* struct to inform when the channel has been connected, disconnected or when the encryption has changed. In addition to that it also contains the `recv` callback which is called whenever an incoming data has been received. Data received this way can be marked as processed by returning 0 or using *bt_l2cap_chan_recv_complete()* API if processing is asynchronous.

Note: The `recv` callback is called directly from RX Thread thus it is not recommended to block for long periods of time.

For sending data the *bt_l2cap_chan_send()* API can be used noting that it may block if no credits are available, and resuming as soon as more credits are available.

Servers can be registered using *bt_l2cap_server_register()* API passing the *bt_l2cap_server* struct which informs what `psm` it should listen to, the required security level `sec_level`, and the callback `accept` which is called to authorize incoming connection requests and allocate channel instances.

Client channels can be initiated with use of *bt_l2cap_chan_connect()* API and can be disconnected with the *bt_l2cap_chan_disconnect()* API. Note that the later can also disconnect channel instances created by servers.

API Reference

group `bt_l2cap`

L2CAP.

Defines

`BT_L2CAP_HDR_SIZE`

L2CAP PDU header size, used for buffer size calculations

`BT_L2CAP_TX_MTU`

Maximum Transmission Unit (MTU) for an outgoing L2CAP PDU.

`BT_L2CAP_RX_MTU`

Maximum Transmission Unit (MTU) for an incoming L2CAP PDU.

`BT_L2CAP_BUF_SIZE(mtu)`

Helper to calculate needed buffer size for L2CAP PDUs. Useful for creating buffer pools.

Parameters

- `mtu` – Needed L2CAP PDU MTU.

Returns Needed buffer size to match the requested L2CAP PDU MTU.

`BT_L2CAP_SDU_HDR_SIZE`

L2CAP SDU header size, used for buffer size calculations

`BT_L2CAP_SDU_TX_MTU`

Maximum Transmission Unit for an unsegmented outgoing L2CAP SDU.

The Maximum Transmission Unit for an outgoing L2CAP SDU when sent without segmentation, i.e a single L2CAP SDU will fit inside a single L2CAP PDU.

The MTU for outgoing L2CAP SDUs with segmentation is defined by the size of the application buffer pool.

`BT_L2CAP_SDU_RX_MTU`

Maximum Transmission Unit for an unsegmented incoming L2CAP SDU.

The Maximum Transmission Unit for an incoming L2CAP SDU when sent without segmentation, i.e a single L2CAP SDU will fit inside a single L2CAP PDU.

The MTU for incoming L2CAP SDUs with segmentation is defined by the size of the application buffer pool. The application will have to define an alloc_buf callback for the channel in order to support receiving segmented L2CAP SDUs.

`BT_L2CAP_SDU_BUF_SIZE(mtu)`

Helper to calculate needed buffer size for L2CAP SDUs. Useful for creating buffer pools.

Parameters

- `mtu` – Required BT_L2CAP_*_SDU.

Returns Needed buffer size to match the requested L2CAP SDU MTU.

`BT_L2CAP_LE_CHAN(_ch)`

Helper macro getting container object of type *bt_l2cap_le_chan* address having the same container chan member address as object in question.

Parameters

- `_ch` – Address of object of *bt_l2cap_chan* type

Returns Address of in memory *bt_l2cap_le_chan* object type containing the address of in question object.

`BT_L2CAP_CHAN_SEND_RESERVE`

Headroom needed for outgoing L2CAP PDUs.

`BT_L2CAP_SDU_CHAN_SEND_RESERVE`

Headroom needed for outgoing L2CAP SDUs.

Typedefs

typedef void (*`bt_l2cap_chan_destroy_t`)(struct *bt_l2cap_chan* *chan)

Channel destroy callback.

Param chan Channel object.

typedef enum *bt_l2cap_chan_state* `bt_l2cap_chan_state_t`

Life-span states of L2CAP CoC channel.

Used only by internal APIs dealing with setting channel to proper state depending on operational context.

typedef enum *bt_l2cap_chan_status* `bt_l2cap_chan_status_t`

Status of L2CAP channel.

Enums

enum `bt_l2cap_chan_state`

Life-span states of L2CAP CoC channel.

Used only by internal APIs dealing with setting channel to proper state depending on operational context.

Values:

enumerator `BT_L2CAP_DISCONNECTED`

Channel disconnected

enumerator `BT_L2CAP_CONNECT`

Channel in connecting state

enumerator `BT_L2CAP_CONFIG`

Channel in config state, BR/EDR specific

enumerator `BT_L2CAP_CONNECTED`

Channel ready for upper layer traffic on it

enumerator `BT_L2CAP_DISCONNECT`

Channel in disconnecting state

enum `bt_l2cap_chan_status`

Status of L2CAP channel.

Values:

enumerator `BT_L2CAP_STATUS_OUT`

Channel output status

enumerator `BT_L2CAP_STATUS_SHUTDOWN`

Channel shutdown status.

```
Once this status is notified it means the channel will no longer be
able to transmit or receive data.
```

enumerator `BT_L2CAP_STATUS_ENCRYPT_PENDING`

Channel encryption pending status.

enumerator `BT_L2CAP_NUM_STATUS`

Functions

int `bt_l2cap_server_register`(struct *bt_l2cap_server* *server)

Register L2CAP server.

Register L2CAP server for a PSM, each new connection is authorized using the accept() callback which in case of success shall allocate the channel structure to be used by the new connection.

For fixed, SIG-assigned PSMs (in the range 0x0001-0x007f) the PSM should be assigned to server->psm before calling this API. For dynamic PSMs (in the range 0x0080-0x00ff) server->psm may be pre-set to a given value (this is however not recommended) or be left as 0, in which case upon return a newly allocated value will have been assigned to it. For dynamically allocated values the expectation is that it's exposed through a GATT service, and that's how L2CAP clients discover how to connect to the server.

Parameters

- `server` – Server structure.

Returns 0 in case of success or negative value in case of error.

int `bt_l2cap_br_server_register`(struct *bt_l2cap_server* *server)

Register L2CAP server on BR/EDR oriented connection.

Register L2CAP server for a PSM, each new connection is authorized using the accept() callback which in case of success shall allocate the channel structure to be used by the new connection.

Parameters

- `server` – Server structure.

Returns 0 in case of success or negative value in case of error.

int `bt_l2cap_ecred_chan_connect`(struct bt_conn *conn, struct *bt_l2cap_chan* **chans, uint16_t psm)

Connect Enhanced Credit Based L2CAP channels.

Connect up to 5 L2CAP channels by PSM, once the connection is completed each channel connected() callback will be called. If the connection is rejected disconnected() callback is called instead.

Parameters

- `conn` – Connection object.
- `chans` – Array of channel objects.
- `psm` – Channel PSM to connect to.

Returns 0 in case of success or negative value in case of error.

int `bt_l2cap_ecred_chan_reconfigure`(struct *bt_l2cap_chan* **chans, uint16_t mtu)

Reconfigure Enhanced Credit Based L2CAP channels.

Reconfigure up to 5 L2CAP channels. Channels must be from the same bt_conn. Once reconfiguration is completed each channel reconfigured() callback will be called. MTU cannot be decreased on any of provided channels.

Parameters

- `chans` – Array of channel objects. Null-terminated. Elements after the first 5 are silently ignored.
- `mtu` – Channel MTU to reconfigure to.

Returns 0 in case of success or negative value in case of error.

int `bt_l2cap_chan_connect`(struct bt_conn *conn, struct *bt_l2cap_chan* *chan, uint16_t psm)

Connect L2CAP channel.

Connect L2CAP channel by PSM, once the connection is completed channel connected() callback will be called. If the connection is rejected disconnected() callback is called instead. Channel object passed (over an address of it) as second parameter shouldn't be instantiated in application as standalone. Instead of, application should create transport dedicated L2CAP objects, i.e. type of *bt_l2cap_le_chan* for LE and/or type of *bt_l2cap_br_chan* for BR/EDR. Then pass to this API the location (address) of *bt_l2cap_chan* type object which is a member of both transport dedicated objects.

Parameters

- `conn` – Connection object.
- `chan` – Channel object.
- `psm` – Channel PSM to connect to.

Returns 0 in case of success or negative value in case of error.

int `bt_l2cap_chan_disconnect`(struct *bt_l2cap_chan* *chan)

Disconnect L2CAP channel.

Disconnect L2CAP channel, if the connection is pending it will be canceled and as a result the channel disconnected() callback is called. Regarding to input parameter, to get details see reference description to *bt_l2cap_chan_connect()* API above.

Parameters

- `chan` – Channel object.

Returns 0 in case of success or negative value in case of error.

int `bt_l2cap_chan_send`(struct *bt_l2cap_chan* *chan, struct *net_buf* *buf)

Send data to L2CAP channel.

Send data from buffer to the channel. If credits are not available, buf will be queued and sent as and when credits are received from peer. Regarding to first input parameter, to get details see reference description to *bt_l2cap_chan_connect()* API above.

When sending L2CAP data over an BR/EDR connection the application is sending L2CAP PDUs. The application is required to have reserved *BT_L2CAP_CHAN_SEND_RESERVE* bytes in the buffer before sending. The application should use the *BT_L2CAP_BUF_SIZE()* helper to correctly size the buffers for the for the outgoing buffer pool.

When sending L2CAP data over an LE connection the applicatios is sending L2CAP SDUs. The application can optionally reserve *BT_L2CAP_SDU_CHAN_SEND_RESERVE* bytes in the buffer before sending. By reserving bytes in the buffer the stack can use this buffer as a segment directly, otherwise it will have to allocate a new segment for the first segment. If the application is reserving the bytes it should use the *BT_L2CAP_BUF_SIZE()* helper to correctly size the buffers for the for the outgoing buffer pool. When segmenting an L2CAP SDU into L2CAP PDUs the stack will first attempt to allocate buffers from the original buffer pool of the L2CAP SDU before using the stacks own buffer pool.

Note: Buffer ownership is transferred to the stack in case of success, in case of an error the caller retains the ownership of the buffer.

Returns Bytes sent in case of success or negative value in case of error.

int bt_l2cap_chan_recv_complete(struct *bt_l2cap_chan* *chan, struct *net_buf* *buf)

Complete receiving L2CAP channel data.

Complete the reception of incoming data. This shall only be called if the channel recv callback has returned -EINPROGRESS to process some incoming data. The buffer shall contain the original user_data as that is used for storing the credits/segments used by the packet.

Parameters

- chan – Channel object.
- buf – Buffer containing the data.

Returns 0 in case of success or negative value in case of error.

struct bt_l2cap_chan

#include <l2cap.h> L2CAP Channel structure.

Public Members

struct bt_conn *conn

Channel connection reference

const struct *bt_l2cap_chan_ops* *ops

Channel operations reference

struct bt_l2cap_le_endpoint

#include <l2cap.h> LE L2CAP Endpoint structure.

Public Members

uint16_t cid

Endpoint Channel Identifier (CID)

uint16_t mtu

Endpoint Maximum Transmission Unit

uint16_t mps

Endpoint Maximum PDU payload Size

uint16_t init_credits

Endpoint initial credits

atomic_t credits

Endpoint credits

struct bt_l2cap_le_chan

#include <l2cap.h> LE L2CAP Channel structure.

Public Members

struct *bt_l2cap_chan* `chan`

Common L2CAP channel reference object

struct *bt_l2cap_le_endpoint* `rx`

Channel Receiving Endpoint.

If the application has set an alloc_buf channel callback for the channel to support receiving segmented L2CAP SDUs the application should inititalize the MTU of the Receiving Endpoint. Otherwise the MTU of the receiving endpoint will be initialized to *BT_L2CAP_SDU_RX_MTU* by the stack.

uint16_t `pending_rx_mtu`

Pending RX MTU on ECFC reconfigure, used internally by stack

struct *bt_l2cap_le_endpoint* `tx`

Channel Transmission Endpoint

struct k_fifo `tx_queue`

Channel Transmission queue

struct *net_buf* *`tx_buf`

Channel Pending Transmission buffer

struct *k_work* `tx_work`

Channel Transmission work

struct `bt_l2cap_br_endpoint`

#include <l2cap.h> BREDR L2CAP Endpoint structure.

Public Members

uint16_t `cid`

Endpoint Channel Identifier (CID)

uint16_t `mtu`

Endpoint Maximum Transmission Unit

struct `bt_l2cap_br_chan`

#include <l2cap.h> BREDR L2CAP Channel structure.

Public Members

struct *bt_l2cap_chan* `chan`

Common L2CAP channel reference object

struct *bt_l2cap_br_endpoint* `rx`

Channel Receiving Endpoint

struct *bt_l2cap_br_endpoint* `tx`

Channel Transmission Endpoint

struct `bt_l2cap_chan_ops`

#include <l2cap.h> L2CAP Channel operations structure.

Public Members

void (*`connected`)(struct *bt_l2cap_chan* *chan)

Channel connected callback.

If this callback is provided it will be called whenever the connection completes.

Param chan The channel that has been connected

void (*`disconnected`)(struct *bt_l2cap_chan* *chan)

Channel disconnected callback.

If this callback is provided it will be called whenever the channel is disconnected, including when a connection gets rejected.

Param chan The channel that has been Disconnected

void (*`encrypt_change`)(struct *bt_l2cap_chan* *chan, uint8_t hci_status)

Channel encrypt_change callback.

If this callback is provided it will be called whenever the security level changed (indirectly link encryption done) or authentication procedure fails. In both cases security initiator and responder got the final status (HCI status) passed by related to encryption and authentication events from local host's controller.

Param chan The channel which has made encryption status changed.

Param status HCI status of performed security procedure caused by channel security requirements. The value is populated by HCI layer and set to 0 when success and to non-zero (reference to HCI Error Codes) when security/authentication failed.

struct *net_buf* *(*`alloc_buf`)(struct *bt_l2cap_chan* *chan)

Channel alloc_buf callback.

If this callback is provided the channel will use it to allocate buffers to store incoming data. Channels that requires segmentation must set this callback. If the application has not set a callback the L2CAP SDU MTU will be truncated to *BT_L2CAP_SDU_RX_MTU*.

Param chan The channel requesting a buffer.

Return Allocated buffer.

int (*`recv`)(struct *bt_l2cap_chan* *chan, struct *net_buf* *buf)

Channel recv callback.

Param chan The channel receiving data.

Param buf Buffer containing incoming data.

Return 0 in case of success or negative value in case of error.

Return -EINPROGRESS in case where user has to confirm once the data has been processed by calling *bt_l2cap_chan_recv_complete* passing back the buffer

received with its original user_data which contains the number of segments/credits used by the packet.

void (*sent)(struct *bt_l2cap_chan* *chan)

Channel sent callback.

If this callback is provided it will be called whenever a SDU has been completely sent.

Param chan The channel which has sent data.

void (*status)(struct *bt_l2cap_chan* *chan, atomic_t *status)

Channel status callback.

If this callback is provided it will be called whenever the channel status changes.

Param chan The channel which status changed

Param status The channel status

void (*reconfigured)(struct *bt_l2cap_chan* *chan)

Channel reconfigured callback.

If this callback is provided it will be called whenever peer or local device requested reconfiguration. Application may check updated MTU and MPS values by inspecting chan->le endpoints.

Param chan The channel which was reconfigured

struct bt_l2cap_server

#include <l2cap.h> L2CAP Server structure.

Public Members

uint16_t psm

Server PSM.

Possible values: 0 A dynamic value will be auto-allocated when *bt_l2cap_server_register()* is called.

0x0001-0x007f Standard, Bluetooth SIG-assigned fixed values.

0x0080-0x00ff Dynamically allocated. May be pre-set by the application before server registration (not recommended however), or auto-allocated by the stack if the app gave 0 as the value.

bt_security_t sec_level

Required minimum security level

int (*accept)(struct bt_conn *conn, struct *bt_l2cap_chan* **chan)

Server accept callback.

This callback is called whenever a new incoming connection requires authorization.

Param conn The connection that is requesting authorization

Param chan Pointer to received the allocated channel

Return 0 in case of success or negative value in case of error.

Return -ENOMEM if no available space for new channel.

Return -EACCES if application did not authorize the connection.

Return -EPERM if encryption key size is too short.

7.4.11 Bluetooth Mesh Profile

The Bluetooth mesh profile adds secure wireless multi-hop communication for Bluetooth Low Energy. This module implements the Bluetooth Mesh Profile Specification v1.0.1.

Read more about Bluetooth mesh on the Bluetooth SIG Website.

Core

The core provides functionality for managing the general Bluetooth mesh state.

Low Power Node The Low Power Node (LPN) role allows battery powered devices to participate in a mesh network as a leaf node. An LPN interacts with the mesh network through a Friend node, which is responsible for relaying any messages directed to the LPN. The LPN saves power by keeping its radio turned off, and only wakes up to either send messages or poll the Friend node for any incoming messages.

The radio control and polling is managed automatically by the mesh stack, but the LPN API allows the application to trigger the polling at any time through *bt_mesh_lpn_poll()*. The LPN operation parameters, including poll interval, poll event timing and Friend requirements is controlled through the CONFIG_BT_MESH_LOW_POWER option and related configuration options.

Replay Protection List The Replay Protection List (RPL) is used to hold recently received sequence numbers from elements within the mesh network to perform protection against replay attacks.

To keep a node protected against replay attacks after reboot, it needs to store the entire RPL in the persistent storage before it is powered off. Depending on the amount of traffic in a mesh network, storing recently seen sequence numbers can make flash wear out sooner or later. To mitigate this, @ref CONFIG_BT_MESH_RPL_STORE_TIMEOUT can be used. This option postpones storing of RPL entries in the persistent storage.

This option, however, doesn't completely solve the issue as the node may get powered off before the timer to store the RPL is fired. To ensure that messages can not be replayed, the node can initiate storage of the pending RPL entry (or entries) at any time (or sufficiently before power loss) by calling @ref bt_mesh_rpl_pending_store. This is up to the node to decide, which RPL entries are to be stored in this case.

Setting @ref CONFIG_BT_MESH_RPL_STORE_TIMEOUT to -1 allows to completely switch off the timer, which can help to significantly reduce flash wear out. This moves the responsibility of storing RPL to the user application and requires that sufficient power backup is available from the time this API is called until all RPL entries are written to the flash.

Finding the right balance between @ref CONFIG_BT_MESH_RPL_STORE_TIMEOUT and calling @ref bt_mesh_rpl_pending_store may reduce a risk of security volnurability and flash wear out.

API reference

group bt_mesh

Bluetooth mesh.

Defines

BT_MESH_NET_PRIMARY

BT_MESH_FEAT_RELAY

Relay feature

BT_MESH_FEAT_PROXY

GATT Proxy feature

BT_MESH_FEAT_FRIEND

Friend feature

BT_MESH_FEAT_LOW_POWER

Low Power Node feature

BT_MESH_FEAT_SUPPORTED

BT_MESH_LPN_CB_DEFINE(_name)

Register a callback structure for Friendship events.

Parameters

- _name – Name of callback structure.

BT_MESH_FRIEND_CB_DEFINE(_name)

Register a callback structure for Friendship events.

Registers a callback structure that will be called whenever Friendship gets established or terminated.

Parameters

- _name – Name of callback structure.

Functions

int bt_mesh_init(const struct *bt_mesh_prov* *prov, const struct *bt_mesh_comp* *comp)

Initialize Mesh support.

After calling this API, the node will not automatically advertise as unprovisioned, rather the *bt_mesh_prov_enable()* API needs to be called to enable unprovisioned advertising on one or more provisioning bearers.

Parameters

- prov – Node provisioning information.
- comp – Node Composition.

Returns Zero on success or (negative) error code otherwise.

void bt_mesh_reset(void)

Reset the state of the local Mesh node.

Resets the state of the node, which means that it needs to be reprovisioned to become an active node in a Mesh network again.

After calling this API, the node will not automatically advertise as unprovisioned, rather the *bt_mesh_prov_enable()* API needs to be called to enable unprovisioned advertising on one or more provisioning bearers.

int bt_mesh_suspend(void)

Suspend the Mesh network temporarily.

This API can be used for power saving purposes, but the user should be aware that leaving the local node suspended for a long period of time may cause it to become permanently disconnected from the Mesh network. If at all possible, the Friendship feature should be used instead, to make the node into a Low Power Node.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_resume`(void)

Resume a suspended Mesh network.

This API resumes the local node, after it has been suspended using the *bt_mesh_suspend()* API.

Returns 0 on success, or (negative) error code on failure.

void `bt_mesh_iv_update_test`(bool enable)

Toggle the IV Update test mode.

This API is only available if the IV Update test mode has been enabled in Kconfig. It is needed for passing most of the IV Update qualification test cases.

Parameters

- `enable` – true to enable IV Update test mode, false to disable it.

bool `bt_mesh_iv_update`(void)

Toggle the IV Update state.

This API is only available if the IV Update test mode has been enabled in Kconfig. It is needed for passing most of the IV Update qualification test cases.

Returns true if IV Update In Progress state was entered, false otherwise.

int `bt_mesh_lpn_set`(bool enable)

Toggle the Low Power feature of the local device.

Enables or disables the Low Power feature of the local device. This is exposed as a run-time feature, since the device might want to change this e.g. based on being plugged into a stable power source or running from a battery power source.

Parameters

- `enable` – true to enable LPN functionality, false to disable it.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_lpn_poll`(void)

Send out a Friend Poll message.

Send a Friend Poll message to the Friend of this node. If there is no established Friendship the function will return an error.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_friend_terminate`(uint16_t lpn_addr)

Terminate Friendship.

Terminated Friendship for given LPN.

Parameters

- `lpn_addr` – Low Power Node address.

Returns Zero on success or (negative) error code otherwise.

void `bt_mesh_rpl_pending_store`(uint16_t addr)

Store pending RPL entry(ies) in the persistent storage.

This API allows the user to store pending RPL entry(ies) in the persistent storage without waiting for the timeout.

Note: When flash is used as the persistent storage, calling this API too frequently may wear it out.

Parameters

- `addr` – Address of the node which RPL entry needs to be stored or *BT_MESH_ADDR_ALL_NODES* to store all pending RPL entries.

struct `bt_mesh_lpn_cb`

#include <main.h> Low Power Node callback functions.

Public Members

void (*`established`)(uint16_t net_idx, uint16_t friend_addr, uint8_t queue_size, uint8_t recv_window)

Friendship established.

This callback notifies the application that friendship has been successfully established.
Param net_idx NetKeyIndex used during friendship establishment.
Param friend_addr Friend address.
Param queue_size Friend queue size.
Param recv_window Low Power Node's listens duration for Friend response.

void (*`terminated`)(uint16_t net_idx, uint16_t friend_addr)

Friendship terminated.

This callback notifies the application that friendship has been terminated.
Param net_idx NetKeyIndex used during friendship establishment.
Param friend_addr Friend address.

void (*`polled`)(uint16_t net_idx, uint16_t friend_addr, bool retry)

Local Poll Request.

This callback notifies the application that the local node has polled the friend node.

This callback will be called before *bt_mesh_lpn_cb::established* when attempting to establish a friendship.
Param net_idx NetKeyIndex used during friendship establishment.
Param friend_addr Friend address.
Param retry Retry or first poll request for each transaction.

struct `bt_mesh_friend_cb`

#include <main.h> Friend Node callback functions.

Public Members

void (*`established`)(uint16_t net_idx, uint16_t lpn_addr, uint8_t recv_delay, uint32_t polltimeout)

Friendship established.

This callback notifies the application that friendship has been successfully established.
Param net_idx NetKeyIndex used during friendship establishment.
Param lpn_addr Low Power Node address.
Param recv_delay Receive Delay in units of 1 millisecond.
Param polltimeout PollTimeout in units of 1 millisecond.

void (*terminated)(uint16_t net_idx, uint16_t lpn_addr)

Friendship terminated.

This callback notifies the application that friendship has been terminated.

Param net_idx NetKeyIndex used during friendship establishment.
Param lpn_addr Low Power Node address.

void (*polled)(uint16_t net_idx, uint16_t lpn_addr)

Friend Poll Request.

This callback notifies the application that the low power node has polled the friend node.

This callback will be called before *bt_mesh_friend_cb::established* when attempting to establish a friendship.

Param net_idx NetKeyIndex used during friendship establishment.
Param lpn_addr LPN address.

Access layer

The access layer is the application's interface to the Bluetooth mesh network. The access layer provides mechanisms for compartmentalizing the node behavior into elements and models, which are implemented by the application.

Mesh models The functionality of a mesh node is represented by models. A model implements a single behavior the node supports, like being a light, a sensor or a thermostat. The mesh models are grouped into *elements*. Each element is assigned its own unicast address, and may only contain one of each type of model. Conventionally, each element represents a single aspect of the mesh node behavior. For instance, a node that contains a sensor, two lights and a power outlet would spread this functionality across four elements, with each element instantiating all the models required for a single aspect of the supported behavior.

The node's element and model structure is specified in the node composition data, which is passed to *bt_mesh_init()* during initialization. The Bluetooth SIG have defined a set of foundation models (see *Foundation models*) and a set of models for implementing common behavior in the Bluetooth Mesh Model Specification. All models not specified by the Bluetooth SIG are vendor models, and must be tied to a Company ID.

Mesh models have several parameters that can be configured either through initialization of the mesh stack or with the *Configuration Server*:

Opcode list The opcode list contains all message opcodes the model can receive, as well as the minimum acceptable payload length and the callback to pass them to. Models can support any number of opcodes, but each opcode can only be listed by one model in each element.

The full opcode list must be passed to the model structure in the composition data, and cannot be changed at runtime. The end of the opcode list is determined by the special *BT_MESH_MODEL_OP_END* entry. This entry must always be present in the opcode list, unless the list is empty. In that case, *BT_MESH_MODEL_NO_OPS* should be used in place of a proper opcode list definition.

AppKey list The AppKey list contains all the application keys the model can receive messages on. Only messages encrypted with application keys in the AppKey list will be passed to the model.

The maximum number of supported application keys each model can hold is configured with the CONFIG_BT_MESH_MODEL_KEY_COUNT configuration option. The contents of the AppKey list is managed by the *Configuration Server*.

Subscription list A model will process all messages addressed to the unicast address of their element (given that the utilized application key is present in the AppKey list). Additionally, the model will process packets addressed to any group or virtual address in its subscription list. This allows nodes to address multiple nodes throughout the mesh network with a single message.

The maximum number of supported addresses in the Subscription list each model can hold is configured with the `CONFIG_BT_MESH_MODEL_GROUP_COUNT` configuration option. The contents of the subscription list is managed by the *Configuration Server*.

Model publication The models may send messages in two ways:

- By specifying a set of message parameters in a *bt_mesh_msg_ctx*, and calling *bt_mesh_model_send()*.
- By setting up a *bt_mesh_model_pub* structure and calling *bt_mesh_model_publish()*.

When publishing messages with *bt_mesh_model_publish()*, the model will use the publication parameters configured by the *Configuration Server*. This is the recommended way to send unprompted model messages, as it passes the responsibility of selecting message parameters to the network administrator, which likely knows more about the mesh network than the individual nodes will.

To support publishing with the publication parameters, the model must allocate a packet buffer for publishing, and pass it to *bt_mesh_model_pub.msg*. The Config Server may also set up period publication for the publication message. To support this, the model must populate the *bt_mesh_model_pub.update* callback. The *bt_mesh_model_pub.update* callback will be called right before the message is published, allowing the model to change the payload to reflect its current state.

Extended models The Bluetooth mesh specification allows the mesh models to extend each other. When a model extends another, it inherits that model's functionality, and extension can be used to construct complex models out of simple ones, leveraging the existing model functionality to avoid defining new opcodes. Models may extend any number of models, from any element. When one model extends another in the same element, the two models will share subscription lists. The mesh stack implements this by merging the subscription lists of the two models into one, combining the number of subscriptions the models can have in total. Models may extend models that extend others, creating an "extension tree". All models in an extension tree share a single subscription list per element it spans.

Model extensions are done by calling *bt_mesh_model_extend()* during initialization. A model can only be extended by one other model, and extensions cannot be circular. Note that binding of node states and other relationships between the models must be defined by the model implementations.

The model extension concept adds some overhead in the access layer packet processing, and must be explicitly enabled with `CONFIG_BT_MESH_MODEL_EXTENSIONS` to have any effect.

Model data storage Mesh models may have data associated with each model instance that needs to be stored persistently. The access API provides a mechanism for storing this data, leveraging the internal model instance encoding scheme. Models can store one user defined data entry per instance by calling *bt_mesh_model_data_store()*. To be able to read out the data the next time the device reboots, the model's *bt_mesh_model_cb.settings_set* callback must be populated. This callback gets called when model specific data is found in the persistent storage. The model can retrieve the data by calling the `read_cb` passed as a parameter to the callback. See the *Settings* module documentation for details.

API reference

group `bt_mesh_access`

Access layer.

Defines

BT_MESH_ADDR_UNASSIGNED

BT_MESH_ADDR_ALL_NODES

BT_MESH_ADDR_PROXIES

BT_MESH_ADDR_FRIENDS

BT_MESH_ADDR_RELAYS

BT_MESH_KEY_UNUSED

BT_MESH_KEY_ANY

BT_MESH_KEY_DEV

BT_MESH_KEY_DEV_LOCAL

BT_MESH_KEY_DEV_REMOTE

BT_MESH_KEY_DEV_ANY

BT_MESH_ADDR_IS_UNICAST(addr)

BT_MESH_ADDR_IS_GROUP(addr)

BT_MESH_ADDR_IS_VIRTUAL(addr)

BT_MESH_ADDR_IS_RFU(addr)

BT_MESH_IS_DEV_KEY(key)

BT_MESH_APP_SEG_SDU_MAX

Maximum payload size of an access message (in octets).

BT_MESH_TX_SDU_MAX

Maximum possible payload size of an outgoing access message (in octets).

BT_MESH_RX_SDU_MAX

Maximum possible payload size of an incoming access message (in octets).

BT_MESH_ELEM(_loc, _mods, _vnd_mods)

Helper to define a mesh element within an array.

In case the element has no SIG or Vendor models the helper macro BT_MESH_MODEL_NONE can be given instead.

Parameters

- `_loc` – Location Descriptor.
- `_mods` – Array of models.

- `_vnd_mods` – Array of vendor models.

`BT_MESH_MODEL_ID_CFG_SRV`

`BT_MESH_MODEL_ID_CFG_CLI`

`BT_MESH_MODEL_ID_HEALTH_SRV`

`BT_MESH_MODEL_ID_HEALTH_CLI`

`BT_MESH_MODEL_ID_GEN_ONOFF_SRV`

`BT_MESH_MODEL_ID_GEN_ONOFF_CLI`

`BT_MESH_MODEL_ID_GEN_LEVEL_SRV`

`BT_MESH_MODEL_ID_GEN_LEVEL_CLI`

`BT_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_SRV`

`BT_MESH_MODEL_ID_GEN_DEF_TRANS_TIME_CLI`

`BT_MESH_MODEL_ID_GEN_POWER_ONOFF_SRV`

`BT_MESH_MODEL_ID_GEN_POWER_ONOFF_SETUP_SRV`

`BT_MESH_MODEL_ID_GEN_POWER_ONOFF_CLI`

`BT_MESH_MODEL_ID_GEN_POWER_LEVEL_SRV`

`BT_MESH_MODEL_ID_GEN_POWER_LEVEL_SETUP_SRV`

`BT_MESH_MODEL_ID_GEN_POWER_LEVEL_CLI`

`BT_MESH_MODEL_ID_GEN_BATTERY_SRV`

`BT_MESH_MODEL_ID_GEN_BATTERY_CLI`

`BT_MESH_MODEL_ID_GEN_LOCATION_SRV`

`BT_MESH_MODEL_ID_GEN_LOCATION_SETUPSRV`

`BT_MESH_MODEL_ID_GEN_LOCATION_CLI`

`BT_MESH_MODEL_ID_GEN_ADMIN_PROP_SRV`

BT_MESH_MODEL_ID_GEN_MANUFACTURER_PROP_SRV

BT_MESH_MODEL_ID_GEN_USER_PROP_SRV

BT_MESH_MODEL_ID_GEN_CLIENT_PROP_SRV

BT_MESH_MODEL_ID_GEN_PROP_CLI

BT_MESH_MODEL_ID_SENSOR_SRV

BT_MESH_MODEL_ID_SENSOR_SETUP_SRV

BT_MESH_MODEL_ID_SENSOR_CLI

BT_MESH_MODEL_ID_TIME_SRV

BT_MESH_MODEL_ID_TIME_SETUP_SRV

BT_MESH_MODEL_ID_TIME_CLI

BT_MESH_MODEL_ID_SCENE_SRV

BT_MESH_MODEL_ID_SCENE_SETUP_SRV

BT_MESH_MODEL_ID_SCENE_CLI

BT_MESH_MODEL_ID_SCHEDULER_SRV

BT_MESH_MODEL_ID_SCHEDULER_SETUP_SRV

BT_MESH_MODEL_ID_SCHEDULER_CLI

BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV

BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_SETUP_SRV

BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_CLI

BT_MESH_MODEL_ID_LIGHT_CTL_SRV

BT_MESH_MODEL_ID_LIGHT_CTL_SETUP_SRV

BT_MESH_MODEL_ID_LIGHT_CTL_CLI

BT_MESH_MODEL_ID_LIGHT_CTL_TEMP_SRV

BT_MESH_MODEL_ID_LIGHT_HSL_SRV

BT_MESH_MODEL_ID_LIGHT_HSL_SETUP_SRV

BT_MESH_MODEL_ID_LIGHT_HSL_CLI

BT_MESH_MODEL_ID_LIGHT_HSL_HUE_SRV

BT_MESH_MODEL_ID_LIGHT_HSL_SAT_SRV

BT_MESH_MODEL_ID_LIGHT_XYL_SRV

BT_MESH_MODEL_ID_LIGHT_XYL_SETUP_SRV

BT_MESH_MODEL_ID_LIGHT_XYL_CLI

BT_MESH_MODEL_ID_LIGHT_LC_SRV

BT_MESH_MODEL_ID_LIGHT_LC_SETUPSRV

BT_MESH_MODEL_ID_LIGHT_LC_CLI

BT_MESH_MODEL_OP_1(b0)

BT_MESH_MODEL_OP_2(b0, b1)

BT_MESH_MODEL_OP_3(b0, cid)

BT_MESH_LEN_EXACT(len)
: Macro for encoding exact message length for fixed-length messages.

BT_MESH_LEN_MIN(len)
: Macro for encoding minimum message length for variable-length messages.

BT_MESH_MODEL_OP_END
: End of the opcode list. Must always be present.

BT_MESH_MODEL_NO_OPS
: Helper to define an empty opcode list.

BT_MESH_MODEL_NONE
: Helper to define an empty model array

BT_MESH_MODEL_CB(_id, _op, _pub, _user_data, _cb)
: Composition data SIG model entry with callback functions.

  Parameters

  - _id – Model ID.
  - _op – Array of model opcode handlers.
  - _pub – Model publish parameters.

- `_user_data` – User data for the model.
- `_cb` – Callback structure, or NULL to keep no callbacks.

`BT_MESH_MODEL_VND_CB(_company, _id, _op, _pub, _user_data, _cb)`

Composition data vendor model entry with callback functions.

Parameters

- `_company` – Company ID.
- `_id` – Model ID.
- `_op` – Array of model opcode handlers.
- `_pub` – Model publish parameters.
- `_user_data` – User data for the model.
- `_cb` – Callback structure, or NULL to keep no callbacks.

`BT_MESH_MODEL(_id, _op, _pub, _user_data)`

Composition data SIG model entry.

Parameters

- `_id` – Model ID.
- `_op` – Array of model opcode handlers.
- `_pub` – Model publish parameters.
- `_user_data` – User data for the model.

`BT_MESH_MODEL_VND(_company, _id, _op, _pub, _user_data)`

Composition data vendor model entry.

Parameters

- `_company` – Company ID.
- `_id` – Model ID.
- `_op` – Array of model opcode handlers.
- `_pub` – Model publish parameters.
- `_user_data` – User data for the model.

`BT_MESH_TRANSMIT(count, int_ms)`

Encode transmission count & interval steps.

Parameters

- `count` – Number of retransmissions (first transmission is excluded).
- `int_ms` – Interval steps in milliseconds. Must be greater than 0, less than or equal to 320, and a multiple of 10.

Returns Mesh transmit value that can be used e.g. for the default values of the configuration model data.

`BT_MESH_TRANSMIT_COUNT(transmit)`

Decode transmit count from a transmit value.

Parameters

- `transmit` – Encoded transmit count & interval value.

Returns Transmission count (actual transmissions is N + 1).

BT_MESH_TRANSMIT_INT(transmit)

Decode transmit interval from a transmit value.

Parameters

- transmit – Encoded transmit count & interval value.

Returns Transmission interval in milliseconds.

BT_MESH_PUB_TRANSMIT(count, int_ms)

Encode Publish Retransmit count & interval steps.

Parameters

- count – Number of retransmissions (first transmission is excluded).
- int_ms – Interval steps in milliseconds. Must be greater than 0 and a multiple of 50.

Returns Mesh transmit value that can be used e.g. for the default values of the configuration model data.

BT_MESH_PUB_TRANSMIT_COUNT(transmit)

Decode Publish Retransmit count from a given value.

Parameters

- transmit – Encoded Publish Retransmit count & interval value.

Returns Retransmission count (actual transmissions is N + 1).

BT_MESH_PUB_TRANSMIT_INT(transmit)

Decode Publish Retransmit interval from a given value.

Parameters

- transmit – Encoded Publish Retransmit count & interval value.

Returns Transmission interval in milliseconds.

BT_MESH_MODEL_PUB_DEFINE(_name, _update, _msg_len)

Define a model publication context.

Parameters

- _name – Variable name given to the context.
- _update – Optional message update callback (may be NULL).
- _msg_len – Length of the publication message.

BT_MESH_TTL_DEFAULT

Special TTL value to request using configured default TTL

BT_MESH_TTL_MAX

Maximum allowed TTL value

Functions

int bt_mesh_model_send(struct *bt_mesh_model* *model, struct *bt_mesh_msg_ctx* *ctx, struct *net_buf_simple* *msg, const struct *bt_mesh_send_cb* *cb, void *cb_data)

Send an Access Layer message.

Parameters

- model – Mesh (client) Model that the message belongs to.

- `ctx` – Message context, includes keys, TTL, etc.
- `msg` – Access Layer payload (the actual message to be sent).
- `cb` – Optional "message sent" callback.
- `cb_data` – User data to be passed to the callback.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_model_publish`(struct *bt_mesh_model* *model)

Send a model publication message.

Before calling this function, the user needs to ensure that the model publication message (*bt_mesh_model_pub::msg*) contains a valid message to be sent. Note that this API is only to be used for non-period publishing. For periodic publishing the app only needs to make sure that *bt_mesh_model_pub::msg* contains a valid message whenever the *bt_mesh_model_pub::update* callback is called.

Parameters

- `model` – Mesh (client) Model that's publishing the message.

Returns 0 on success, or (negative) error code on failure.

struct *bt_mesh_elem* *`bt_mesh_model_elem`(struct *bt_mesh_model* *mod)

Get the element that a model belongs to.

Parameters

- `mod` – Mesh model.

Returns Pointer to the element that the given model belongs to.

struct *bt_mesh_model* *`bt_mesh_model_find`(const struct *bt_mesh_elem* *elem, uint16_t id)

Find a SIG model.

Parameters

- `elem` – Element to search for the model in.
- `id` – Model ID of the model.

Returns A pointer to the Mesh model matching the given parameters, or NULL if no SIG model with the given ID exists in the given element.

struct *bt_mesh_model* *`bt_mesh_model_find_vnd`(const struct *bt_mesh_elem* *elem, uint16_t company, uint16_t id)

Find a vendor model.

Parameters

- `elem` – Element to search for the model in.
- `company` – Company ID of the model.
- `id` – Model ID of the model.

Returns A pointer to the Mesh model matching the given parameters, or NULL if no vendor model with the given ID exists in the given element.

static inline bool `bt_mesh_model_in_primary`(const struct *bt_mesh_model* *mod)

Get whether the model is in the primary element of the device.

Parameters

- `mod` – Mesh model.

Returns true if the model is on the primary element, false otherwise.

int bt_mesh_model_data_store(struct *bt_mesh_model* *mod, bool vnd, const char *name, const void *data, size_t data_len)

Immediately store the model's user data in persistent storage.

Parameters

- mod – Mesh model.
- vnd – This is a vendor model.
- name – Name/key of the settings item. Only *SETTINGS_MAX_DIR_DEPTH* bytes will be used at most.
- data – Model data to store, or NULL to delete any model data.
- data_len – Length of the model data.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_model_extend(struct *bt_mesh_model* *extending_mod, struct *bt_mesh_model* *base_mod)

Let a model extend another.

Mesh models may be extended to reuse their functionality, forming a more complex model. A Mesh model may extend any number of models, in any element. The extensions may also be nested, ie a model that extends another may itself be extended.

A set of models that extend each other form a model extension list.

All models in an extension list share one subscription list per element. The access layer will utilize the combined subscription list of all models in an extension list and element, giving the models extended subscription list capacity.

Parameters

- extending_mod – Mesh model that is extending the base model.
- base_mod – The model being extended.

Return values 0 – Successfully extended the base_mod model.

bool bt_mesh_model_is_extended(struct *bt_mesh_model* *model)

Check if model is extended by another model.

Parameters

- model – The model to check.

Return values true – If model is extended by another model, otherwise false

struct bt_mesh_elem

#include <access.h> Abstraction that describes a Mesh Element

Public Members

uint16_t addr

Unicast Address. Set at runtime during provisioning.

const uint16_t loc

Location Descriptor (GATT Bluetooth Namespace Descriptors)

const uint8_t model_count

The number of SIG models in this element

const uint8_t `vnd_model_count`

The number of vendor models in this element

struct *bt_mesh_model* *const `models`

The list of SIG models in this element

struct *bt_mesh_model* *const `vnd_models`

The list of vendor models in this element

struct `bt_mesh_model_op`

#include <access.h> Model opcode handler.

Public Members

const uint32_t `opcode`

OpCode encoded using the BT_MESH_MODEL_OP_* macros

const ssize_t `len`

Message length. If the message has variable length then this value indicates minimum message length and should be positive. Handler function should verify precise length based on the contents of the message. If the message has fixed length then this value should be negative. Use BT_MESH_LEN_* macros when defining this value.

int (*const `func`)(struct *bt_mesh_model* *model, struct *bt_mesh_msg_ctx* *ctx, struct *net_buf_simple* *buf)

Handler function for this opcode.

Param model Model instance receiving the message.
Param ctx Message context for the message.
Param buf Message buffer containing the message payload, not including the opcode.
Return Zero on success or (negative) error code otherwise.

struct `bt_mesh_model_pub`

#include <access.h> Model publication context.

The context should primarily be created using the BT_MESH_MODEL_PUB_DEFINE macro.

Public Members

struct *bt_mesh_model* *`mod`

The model the context belongs to. Initialized by the stack.

uint16_t `addr`

Publish Address.

uint16_t `key`

Publish AppKey Index.

uint16_t `cred`

Friendship Credentials Flag.

uint16_t `send_rel`

Force reliable sending (segment acks)

uint16_t `fast_period`

Use FastPeriodDivisor

uint8_t `ttl`

Publish Time to Live.

uint8_t `retransmit`

Retransmit Count & Interval Steps.

uint8_t `period`

Publish Period.

uint8_t `period_div`

Divisor for the Period.

uint8_t `count`

Transmissions left.

uint32_t `period_start`

Start of the current period.

struct *net_buf_simple* *`msg`

Publication buffer, containing the publication message.

This will get correctly created when the publication context has been defined using the BT_MESH_MODEL_PUB_DEFINE macro.

```
BT_MESH_MODEL_PUB_DEFINE(name, update, size);
```

int (*`update`)(struct *bt_mesh_model* *mod)

Callback for updating the publication buffer.

When set to NULL, the model is assumed not to support periodic publishing. When set to non-NULL the callback will be called periodically and is expected to update *bt_mesh_model_pub::msg* with a valid publication message.

If the callback returns non-zero, the publication is skipped and will resume on the next periodic publishing interval.

Param mod The Model the Publication Context belogs to.
Return Zero on success or (negative) error code otherwise.

struct *k_work_delayable* `timer`

Publish Period Timer. Only for stack-internal use.

struct `bt_mesh_model_cb`

#include <access.h> Model callback functions.

Public Members

int (*const `settings_set`)(struct *bt_mesh_model* *model, const char *name, size_t len_rd, *settings_read_cb* read_cb, void *cb_arg)

Set value handler of user data tied to the model.

See also:

settings_handler::h_set

Param model Model to set the persistent data of.
Param name Name/key of the settings item.
Param len_rd The size of the data found in the backend.
Param read_cb Function provided to read the data from the backend.
Param cb_arg Arguments for the read function provided by the backend.
Return 0 on success, error otherwise.

int (*const `start`)(struct *bt_mesh_model* *model)

Callback called when the mesh is started.

This handler gets called after the node has been provisioned, or after all mesh data has been loaded from persistent storage.

When this callback fires, the mesh model may start its behavior, and all Access APIs are ready for use.

Param model Model this callback belongs to.
Return 0 on success, error otherwise.

int (*const `init`)(struct *bt_mesh_model* *model)

Model init callback.

Called on every model instance during mesh initialization.

If any of the model init callbacks return an error, the Mesh subsystem initialization will be aborted, and the error will be returned to the caller of *bt_mesh_init*.

Param model Model to be initialized.
Return 0 on success, error otherwise.

void (*const `reset`)(struct *bt_mesh_model* *model)

Model reset callback.

Called when the mesh node is reset. All model data is deleted on reset, and the model should clear its state.

Note: If the model stores any persistent data, this needs to be erased manually.

Param model Model this callback belongs to.

struct `bt_mesh_mod_id_vnd`

#include <access.h> Vendor model ID

Public Members

uint16_t `company`
Vendor's company ID

uint16_t `id`
Model ID

struct `bt_mesh_model`
#include <access.h> Abstraction that describes a Mesh Model instance

Public Members

const uint16_t `id`
SIG model ID

const struct *bt_mesh_mod_id_vnd* `vnd`
Vendor model ID

struct *bt_mesh_model_pub* *const `pub`
Model Publication

uint16_t `keys`[CONFIG_BT_MESH_MODEL_KEY_COUNT]
AppKey List

uint16_t `groups`[CONFIG_BT_MESH_MODEL_GROUP_COUNT]
Subscription List (group or virtual addresses)

const struct *bt_mesh_model_op* *const `op`
Opcode handler list

const struct *bt_mesh_model_cb* *const `cb`
Model callback structure.

void *`user_data`
Model-specific user data

struct `bt_mesh_send_cb`
#include <access.h> Callback structure for monitoring model message sending

Public Members

void (*`start`)(uint16_t duration, int err, void *cb_data)
Handler called at the start of the transmission.
Param duration The duration of the full transmission.
Param err Error occurring during sending.
Param cb_data Callback data, as passed to the send API.

void (*end)(int err, void *cb_data)

Handler called at the end of the transmission.

Param err Error occurring during sending.

Param cb_data Callback data, as passed to the send API.

struct bt_mesh_comp

#include <access.h> Node Composition

Public Members

uint16_t cid

Company ID

uint16_t pid

Product ID

uint16_t vid

Version ID

size_t elem_count

The number of elements in this device.

struct *bt_mesh_elem* *elem

List of elements.

Foundation models

The Bluetooth mesh specification defines four foundation models that can be used by network administrators to configure and diagnose mesh nodes.

Configuration Server The Configuration Server model is a foundation model defined by the Bluetooth mesh specification. The Configuration Server model controls most parameters of the mesh node. It does not have an API of its own, but relies on a *Configuration Client* to control it.

..note:: The bt_mesh_cfg_srv structure has been deprecated. The initial values of the Relay, Beacon, Friend, Network transmit and Relay retransmit should be set through Kconfig, and the Heartbeat feature should be controlled through the *Heartbeat* API.

The Configuration Server model is mandatory on all Bluetooth mesh nodes, and should be instantiated in the first element.

API reference

group bt_mesh_cfg_srv

Configuration Server Model.

Defines

BT_MESH_MODEL_CFG_SRV

Generic Configuration Server model composition data entry.

Configuration Client The Configuration Client model is a foundation model defined by the Bluetooth mesh specification. It provides functionality for configuring most parameters of a mesh node, including encryption keys, model configuration and feature enabling.

The Configuration Client model communicates with a *Configuration Server* model using the device key of the target node. The Configuration Client model may communicate with servers on other nodes or self-configure through the local Configuration Server model.

All configuration functions in the Configuration Client API have `net_idx` and `addr` as their first parameters. These should be set to the network index and primary unicast address that the target node was provisioned with.

The Configuration Client model is optional, but should be instantiated on the first element if it is present in the composition data.

API reference

group `bt_mesh_cfg_cli`

Configuration Client Model.

Defines

`BT_MESH_MODEL_CFG_CLI(cli_data)`

Generic Configuration Client model composition data entry.

Parameters

- `cli_data` – Pointer to a *Configuration Client Model* instance.

`BT_MESH_PUB_PERIOD_100MS(steps)`

Helper macro to encode model publication period in units of 100ms.

Parameters

- `steps` – Number of 100ms steps.

Returns Encoded value that can be assigned to *bt_mesh_cfg_mod_pub.period*

`BT_MESH_PUB_PERIOD_SEC(steps)`

Helper macro to encode model publication period in units of 1 second.

Parameters

- `steps` – Number of 1 second steps.

Returns Encoded value that can be assigned to *bt_mesh_cfg_mod_pub.period*

`BT_MESH_PUB_PERIOD_10SEC(steps)`

Helper macro to encode model publication period in units of 10 seconds.

Parameters

- `steps` – Number of 10 second steps.

Returns Encoded value that can be assigned to *bt_mesh_cfg_mod_pub.period*

`BT_MESH_PUB_PERIOD_10MIN(steps)`

Helper macro to encode model publication period in units of 10 minutes.

Parameters

- `steps` – Number of 10 minute steps.

Returns Encoded value that can be assigned to *bt_mesh_cfg_mod_pub.period*

Functions

int `bt_mesh_cfg_node_reset`(uint16_t net_idx, uint16_t addr, bool *status)

Reset the target node and remove it from the network.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `status` – Status response parameter

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_comp_data_get`(uint16_t net_idx, uint16_t addr, uint8_t page, uint8_t *rsp, struct *net_buf_simple* *comp)

Get the target node's composition data.

If the other device does not have the given composition data page, it will return the largest page number it supports that is less than the requested page index. The actual page the device responds with is returned in `rsp`.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `page` – Composition data page, or 0xff to request the first available page.
- `rsp` – Return parameter for the returned page number, or NULL.
- `comp` – Composition data buffer to fill.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_beacon_get`(uint16_t net_idx, uint16_t addr, uint8_t *status)

Get the target node's network beacon state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `status` – Status response parameter, returns one of *BT_MESH_BEACON_DISABLED* or *BT_MESH_BEACON_ENABLED* on success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_krp_get`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status, uint8_t *phase)

Get the target node's network key refresh phase state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index.
- `status` – Status response parameter.
- `phase` – Pointer to the Key Refresh variable to fill.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_krp_set(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t transition, uint8_t *status, uint8_t *phase)

Set the target node's network key refresh phase parameters.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- key_net_idx – Network key index.
- transition – Transition parameter.
- status – Status response parameter.
- phase – Pointer to the new Key Refresh phase. Will return the actual Key Refresh phase after updating.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_beacon_set(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *status)

Set the target node's network beacon state.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- val – New network beacon state, should be one of *BT_MESH_BEACON_DISABLED* or *BT_MESH_BEACON_ENABLED*.
- status – Status response parameter. Returns one of *BT_MESH_BEACON_DISABLED* or *BT_MESH_BEACON_ENABLED* on success.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_ttl_get(uint16_t net_idx, uint16_t addr, uint8_t *ttl)

Get the target node's Time To Live value.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- ttl – TTL response buffer.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_ttl_set(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *ttl)

Set the target node's Time To Live value.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- val – New Time To Live value.
- ttl – TTL response buffer.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_friend_get(uint16_t net_idx, uint16_t addr, uint8_t *status)

Get the target node's Friend feature status.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `status` – Status response parameter. Returns one of *BT_MESH_FRIEND_DISABLED*, *BT_MESH_FRIEND_ENABLED* or *BT_MESH_FRIEND_NOT_SUPPORTED* on success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_friend_set`(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *status)

Set the target node's Friend feature state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – New Friend feature state. Should be one of *BT_MESH_FRIEND_DISABLED* or *BT_MESH_FRIEND_ENABLED*.
- `status` – Status response parameter. Returns one of *BT_MESH_FRIEND_DISABLED*, *BT_MESH_FRIEND_ENABLED* or *BT_MESH_FRIEND_NOT_SUPPORTED* on success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_gatt_proxy_get`(uint16_t net_idx, uint16_t addr, uint8_t *status)

Get the target node's Proxy feature state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `status` – Status response parameter. Returns one of *BT_MESH_GATT_PROXY_DISABLED*, *BT_MESH_GATT_PROXY_ENABLED* or *BT_MESH_GATT_PROXY_NOT_SUPPORTED* on success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_gatt_proxy_set`(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *status)

Set the target node's Proxy feature state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – New Proxy feature state. Must be one of *BT_MESH_GATT_PROXY_DISABLED* or *BT_MESH_GATT_PROXY_ENABLED*.
- `status` – Status response parameter. Returns one of *BT_MESH_GATT_PROXY_DISABLED*, *BT_MESH_GATT_PROXY_ENABLED* or *BT_MESH_GATT_PROXY_NOT_SUPPORTED* on success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_net_transmit_get`(uint16_t net_idx, uint16_t addr, uint8_t *transmit)

Get the target node's network_transmit state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.

- `transmit` – Network transmit response parameter. Returns the encoded network transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_net_transmit_set`(uint16_t net_idx, uint16_t addr, uint8_t val, uint8_t *transmit)

Set the target node's network transmit parameters.

See also:

BT_MESH_TRANSMIT.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `val` – New encoded network transmit parameters.
- `transmit` – Network transmit response parameter. Returns the encoded network transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_relay_get`(uint16_t net_idx, uint16_t addr, uint8_t *status, uint8_t *transmit)

Get the target node's Relay feature state.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `status` – Status response parameter. Returns one of *BT_MESH_RELAY_DISABLED*, *BT_MESH_RELAY_ENABLED* or *BT_MESH_RELAY_NOT_SUPPORTED* on success.
- `transmit` – Transmit response parameter. Returns the encoded relay transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_relay_set`(uint16_t net_idx, uint16_t addr, uint8_t new_relay, uint8_t new_transmit, uint8_t *status, uint8_t *transmit)

Set the target node's Relay parameters.

See also:

BT_MESH_TRANSMIT.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `new_relay` – New relay state. Must be one of *BT_MESH_RELAY_DISABLED* or *BT_MESH_RELAY_ENABLED*.
- `new_transmit` – New encoded relay transmit parameters.

- `status` – Status response parameter. Returns one of *BT_MESH_RELAY_DISABLED*, *BT_MESH_RELAY_ENABLED* or *BT_MESH_RELAY_NOT_SUPPORTED* on success.
- `transmit` – Transmit response parameter. Returns the encoded relay transmission parameters on success. Decoded with *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT*.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_net_key_add`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, const uint8_t net_key[16], uint8_t *status)

Add a network key to the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index.
- `net_key` – Network key.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_net_key_get`(uint16_t net_idx, uint16_t addr, uint16_t *keys, size_t *key_cnt)

Get a list of the target node's network key indexes.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `keys` – Net key index list response parameter. Will be filled with all the returned network key indexes it can fill.
- `key_cnt` – Net key index list length. Should be set to the capacity of the `keys` list when calling. Will return the number of returned network key indexes upon success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_net_key_del`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status)

Delete a network key from the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_app_key_add`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint16_t key_app_idx, const uint8_t app_key[16], uint8_t *status)

Add an application key to the target node.

Parameters

- `net_idx` – Network index to encrypt with.

- addr – Target node address.
- key_net_idx – Network key index the application key belongs to.
- key_app_idx – Application key index.
- app_key – Application key.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_app_key_get(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status, uint16_t *keys, size_t *key_cnt)

Get a list of the target node's application key indexes for a specific network key.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- key_net_idx – Network key index to request the app key indexes of.
- status – Status response parameter.
- keys – App key index list response parameter. Will be filled with all the returned application key indexes it can fill.
- key_cnt – App key index list length. Should be set to the capacity of the keys list when calling. Will return the number of returned application key indexes upon success.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_app_key_del(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint16_t key_app_idx, uint8_t *status)

Delete an application key from the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- key_net_idx – Network key index the application key belongs to.
- key_app_idx – Application key index.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_app_bind(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint8_t *status)

Bind an application to a SIG model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_app_idx – Application index to bind.
- mod_id – Model ID.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_app_unbind(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint8_t *status)

Unbind an application from a SIG model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_app_idx – Application index to unbind.
- mod_id – Model ID.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_app_bind_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint16_t cid, uint8_t *status)

Bind an application to a vendor model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_app_idx – Application index to bind.
- mod_id – Model ID.
- cid – Company ID of the model.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_app_unbind_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_app_idx, uint16_t mod_id, uint16_t cid, uint8_t *status)

Unbind an application from a vendor model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_app_idx – Application index to unbind.
- mod_id – Model ID.
- cid – Company ID of the model.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_app_get(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint8_t *status, uint16_t *apps, size_t *app_cnt)

Get a list of all applications bound to a SIG model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_id – Model ID.
- status – Status response parameter.
- apps – App index list response parameter. Will be filled with all the returned application key indexes it can fill.
- app_cnt – App index list length. Should be set to the capacity of the apps list when calling. Will return the number of returned application key indexes upon success.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_app_get_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, uint8_t *status, uint16_t *apps, size_t *app_cnt)

Get a list of all applications bound to a vendor model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_id – Model ID.
- cid – Company ID of the model.
- status – Status response parameter.
- apps – App index list response parameter. Will be filled with all the returned application key indexes it can fill.
- app_cnt – App index list length. Should be set to the capacity of the apps list when calling. Will return the number of returned application key indexes upon success.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_pub_get(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, struct *bt_mesh_cfg_mod_pub* *pub, uint8_t *status)

Get publish parameters for a SIG model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_id – Model ID.
- pub – Publication parameter return buffer.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_pub_get_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, struct *bt_mesh_cfg_mod_pub* *pub, uint8_t *status)

Get publish parameters for a vendor model on the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `pub` – Publication parameter return buffer.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_mod_pub_set`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, struct *bt_mesh_cfg_mod_pub* *pub, uint8_t *status)

Set publish parameters for a SIG model on the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `mod_id` – Model ID.
- `pub` – Publication parameters.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_mod_pub_set_vnd`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, struct *bt_mesh_cfg_mod_pub* *pub, uint8_t *status)

Set publish parameters for a vendor model on the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `pub` – Publication parameters.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_mod_sub_add`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint8_t *status)

Add a group address to a SIG model's subscription list.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `sub_addr` – Group address to add to the subscription list.

- `mod_id` – Model ID.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_mod_sub_add_vnd`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Add a group address to a vendor model's subscription list.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `sub_addr` – Group address to add to the subscription list.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_mod_sub_del`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint8_t *status)

Delete a group address in a SIG model's subscription list.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `sub_addr` – Group address to add to the subscription list.
- `mod_id` – Model ID.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_mod_sub_del_vnd`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Delete a group address in a vendor model's subscription list.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `sub_addr` – Group address to add to the subscription list.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_overwrite(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint8_t *status)

Overwrite all addresses in a SIG model's subscription list with a group address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- sub_addr – Group address to add to the subscription list.
- mod_id – Model ID.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_overwrite_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t sub_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Overwrite all addresses in a vendor model's subscription list with a group address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- sub_addr – Group address to add to the subscription list.
- mod_id – Model ID.
- cid – Company ID of the model.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_va_add(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t *virt_addr, uint8_t *status)

Add a virtual address to a SIG model's subscription list.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- label – Virtual address label to add to the subscription list.
- mod_id – Model ID.
- virt_addr – Virtual address response parameter.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_va_add_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t cid, uint16_t *virt_addr, uint8_t *status)

Add a virtual address to a vendor model's subscription list.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- label – Virtual address label to add to the subscription list.
- mod_id – Model ID.
- cid – Company ID of the model.
- virt_addr – Virtual address response parameter.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_va_del(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t *virt_addr, uint8_t *status)

Delete a virtual address in a SIG model's subscription list.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- label – Virtual address parameter to add to the subscription list.
- mod_id – Model ID.
- virt_addr – Virtual address response parameter.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_va_del_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t cid, uint16_t *virt_addr, uint8_t *status)

Delete a virtual address in a vendor model's subscription list.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- label – Virtual address label to add to the subscription list.
- mod_id – Model ID.
- cid – Company ID of the model.
- virt_addr – Virtual address response parameter.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_va_overwrite(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t *virt_addr, uint8_t *status)

Overwrite all addresses in a SIG model's subscription list with a virtual address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- label – Virtual address label to add to the subscription list.
- mod_id – Model ID.
- virt_addr – Virtual address response parameter.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_va_overwrite_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, const uint8_t label[16], uint16_t mod_id, uint16_t cid, uint16_t *virt_addr, uint8_t *status)

Overwrite all addresses in a vendor model's subscription list with a virtual address.

Deletes all subscriptions in the model's subscription list, and adds a single group address instead.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- label – Virtual address label to add to the subscription list.
- mod_id – Model ID.
- cid – Company ID of the model.
- virt_addr – Virtual address response parameter.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_get(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint8_t *status, uint16_t *subs, size_t *sub_cnt)

Get the subscription list of a SIG model on the target node.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_id – Model ID.
- status – Status response parameter.

- `subs` – Subscription list response parameter. Will be filled with all the returned subscriptions it can fill.
- `sub_cnt` – Subscription list element count. Should be set to the capacity of the `subs` list when calling. Will return the number of returned subscriptions upon success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_mod_sub_get_vnd`(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, uint8_t *status, uint16_t *subs, size_t *sub_cnt)

Get the subscription list of a vendor model on the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `elem_addr` – Element address the model is in.
- `mod_id` – Model ID.
- `cid` – Company ID of the model.
- `status` – Status response parameter.
- `subs` – Subscription list response parameter. Will be filled with all the returned subscriptions it can fill.
- `sub_cnt` – Subscription list element count. Should be set to the capacity of the `subs` list when calling. Will return the number of returned subscriptions upon success.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_hb_sub_set`(uint16_t net_idx, uint16_t addr, struct *bt_mesh_cfg_hb_sub* *sub, uint8_t *status)

Set the target node's Heartbeat subscription parameters.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `sub` – New Heartbeat subscription parameters.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_hb_sub_get`(uint16_t net_idx, uint16_t addr, struct *bt_mesh_cfg_hb_sub* *sub, uint8_t *status)

Get the target node's Heartbeta subscription parameters.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `sub` – Heartbeat subscription parameter return buffer.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_hb_pub_set(uint16_t net_idx, uint16_t addr, const struct *bt_mesh_cfg_hb_pub* *pub, uint8_t *status)

Set the target node's Heartbeat publication parameters.

Note: The target node must already have received the specified network key.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- pub – New Heartbeat publication parameters.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_hb_pub_get(uint16_t net_idx, uint16_t addr, struct *bt_mesh_cfg_hb_pub* *pub, uint8_t *status)

Get the target node's Heartbeat publication parameters.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- pub – Heartbeat publication parameter return buffer.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_del_all(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint8_t *status)

Delete all group addresses in a SIG model's subscription list.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_id – Model ID.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_cfg_mod_sub_del_all_vnd(uint16_t net_idx, uint16_t addr, uint16_t elem_addr, uint16_t mod_id, uint16_t cid, uint8_t *status)

Delete all group addresses in a vendor model's subscription list.

Parameters

- net_idx – Network index to encrypt with.
- addr – Target node address.
- elem_addr – Element address the model is in.
- mod_id – Model ID.
- cid – Company ID of the model.
- status – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_net_key_update`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, const uint8_t net_key[16], uint8_t *status)

Update a network key to the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index.
- `net_key` – Network key.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_app_key_update`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint16_t key_app_idx, const uint8_t app_key[16], uint8_t *status)

Update an application key to the target node.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `key_net_idx` – Network key index the application key belongs to.
- `key_app_idx` – Application key index.
- `app_key` – Application key.
- `status` – Status response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_node_identity_set`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t new_identity, uint8_t *status, uint8_t *identity)

Set the Node Identity parameters.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `new_identity` – New identity state. Must be one of *BT_MESH_NODE_IDENTITY_STOPPED* or *BT_MESH_NODE_IDENTITY_RUNNING*
- `key_net_idx` – Network key index the application key belongs to.
- `status` – Status response parameter.
- `identity` – Identity response parameter.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_node_identity_get`(uint16_t net_idx, uint16_t addr, uint16_t key_net_idx, uint8_t *status, uint8_t *identity)

Get the Node Identity parameters.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.

- `key_net_idx` – Network key index the application key belongs to.
- `status` – Status response parameter.
- `identity` – Identity response parameter. Must be one of *BT_MESH_NODE_IDENTITY_STOPPED* or *BT_MESH_NODE_IDENTITY_RUNNING*

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_cfg_lpn_timeout_get`(uint16_t net_idx, uint16_t addr, uint16_t unicast_addr, int32_t *polltimeout)

Get the Low Power Node Polltimeout parameters.

Parameters

- `net_idx` – Network index to encrypt with.
- `addr` – Target node address.
- `unicast_addr` – LPN unicast address.
- `polltimeout` – Poltimeout response parameter.

Returns 0 on success, or (negative) error code on failure.

int32_t `bt_mesh_cfg_cli_timeout_get`(void)

Get the current transmission timeout value.

Returns The configured transmission timeout in milliseconds.

void `bt_mesh_cfg_cli_timeout_set`(int32_t timeout)

Set the transmission timeout value.

Parameters

- `timeout` – The new transmission timeout.

int `bt_mesh_comp_p0_get`(struct *bt_mesh_comp_p0* *comp, struct *net_buf_simple* *buf)

Create a composition data page 0 representation from a buffer.

The composition data page object will take ownership over the buffer, which should not be manipulated directly after this call.

This function can be used in combination with *bt_mesh_cfg_comp_data_get* to read out composition data page 0 from other devices:

```
NET_BUF_SIMPLE_DEFINE(buf, BT_MESH_RX_SDU_MAX);
struct bt_mesh_comp_p0 comp;

err = bt_mesh_cfg_comp_data_get(net_idx, addr, 0, &page, &buf);
if (!err) {
        bt_mesh_comp_p0_get(&comp, &buf);
}
```

Parameters

- `buf` – Network buffer containing composition data.
- `comp` – Composition data structure to fill.

Returns 0 on success, or (negative) error code on failure.

struct *bt_mesh_comp_p0_elem* *`bt_mesh_comp_p0_elem_pull`(const struct *bt_mesh_comp_p0* *comp, struct *bt_mesh_comp_p0_elem* *elem)

Pull a composition data page 0 element from a composition data page 0 instance.

Each call to this function will pull out a new element from the composition data page, until all elements have been pulled.

Parameters

- `comp` – Composition data page
- `elem` – Element to fill.

Returns A pointer to `elem` on success, or NULL if no more elements could be pulled.

uint16_t `bt_mesh_comp_p0_elem_mod`(struct *bt_mesh_comp_p0_elem* *elem, int idx)

Get a SIG model from the given composition data page 0 element.

Parameters

- `elem` – Element to read the model from.
- `idx` – Index of the SIG model to read.

Returns The Model ID of the SIG model at the given index, or 0xffff if the index is out of bounds.

struct *bt_mesh_mod_id_vnd* `bt_mesh_comp_p0_elem_mod_vnd`(struct *bt_mesh_comp_p0_elem* *elem, int idx)

Get a vendor model from the given composition data page 0 element.

Parameters

- `elem` – Element to read the model from.
- `idx` – Index of the vendor model to read.

Returns The model ID of the vendor model at the given index, or {0xffff, 0xffff} if the index is out of bounds.

struct `bt_mesh_cfg_cli`

#include <cfg_cli.h> Mesh Configuration Client Model Context

Public Members

struct *bt_mesh_model* *`model`

Composition data model entry pointer.

struct `bt_mesh_cfg_mod_pub`

#include <cfg_cli.h> Model publication configuration parameters.

Public Members

uint16_t `addr`

Publication destination address.

const uint8_t *`uuid`

Virtual address UUID, or NULL if this is not a virtual address.

uint16_t `app_idx`

Application index to publish with.

bool `cred_flag`

Friendship credential flag.

uint8_t `ttl`

Time To Live to publish with.

uint8_t `period`

Encoded publish period.

See also:

BT_MESH_PUB_PERIOD_100MS, *BT_MESH_PUB_PERIOD_SEC*, *BT_MESH_PUB_PERIOD_10SEC*, *BT_MESH_PUB_PERIOD_10MIN*

uint8_t `transmit`

Encoded transmit parameters.

See also:

BT_MESH_TRANSMIT

struct `bt_mesh_cfg_hb_sub`

#include <cfg_cli.h> Heartbeat subscription configuration parameters.

Public Members

uint16_t `src`

Source address to receive Heartbeat messages from.

uint16_t `dst`

Destination address to receive Heartbeat messages on.

uint8_t `period`

Logarithmic subscription period to keep listening for. The decoded subscription period is (1 << (period - 1)) seconds, or 0 seconds if period is 0.

uint8_t `count`

Logarithmic Heartbeat subscription receive count. The decoded Heartbeat count is (1 << (count - 1)) if count is between 1 and 0xfe, 0 if count is 0 and 0xffff if count is 0xff.

Ignored in Heartbeat subscription set.

uint8_t `min`

Minimum hops in received messages, ie the shortest registered path from the publishing node to the subscribing node. A Heartbeat received from an immediate neighbor has hop count = 1.

Ignored in Heartbeat subscription set.

uint8_t `max`

Maximum hops in received messages, ie the longest registered path from the publishing node to the subscribing node. A Heartbeat received from an immediate neighbor has hop count = 1.

Ignored in Heartbeat subscription set.

struct `bt_mesh_cfg_hb_pub`

#include <cfg_cli.h> Heartbeat publication configuration parameters.

Public Members

uint16_t `dst`

Heartbeat destination address.

uint8_t `count`

Logarithmic Heartbeat count. Decoded as (1 << (count - 1)) if count is between 1 and 0x11, 0 if count is 0, or "indefinitely" if count is 0xff.

When used in Heartbeat publication set, this parameter denotes the number of Heartbeat messages to send.

When returned from Heartbeat publication get, this parameter denotes the number of Heartbeat messages remaining to be sent.

uint8_t `period`

Logarithmic Heartbeat publication transmit interval in seconds. Decoded as (1 << (period - 1)) if period is between 1 and 0x11. If period is 0, Heartbeat publication is disabled.

uint8_t `ttl`

Publication message Time To Live value.

uint16_t `feat`

Bitmap of features that trigger Heartbeat publications. Legal values are *BT_MESH_FEAT_RELAY*, *BT_MESH_FEAT_PROXY*, *BT_MESH_FEAT_FRIEND* and *BT_MESH_FEAT_LOW_POWER*

uint16_t `net_idx`

Network index to publish with.

struct `bt_mesh_comp_p0`

#include <cfg_cli.h> Parsed Composition data page 0 representation.

Should be pulled from the return buffer passed to *bt_mesh_cfg_comp_data_get* using *bt_mesh_comp_p0_get*.

Public Members

uint16_t `cid`

Company ID

uint16_t `pid`
Product ID

uint16_t `vid`
Version ID

uint16_t `crpl`
Replay protection list size

uint16_t `feat`
Supported features, see *BT_MESH_FEAT_SUPPORTED*.

struct `bt_mesh_comp_p0_elem`
#include <cfg_cli.h> Composition data page 0 element representation

Public Members

uint16_t `loc`
Element location

size_t `nsig`
The number of SIG models in this element

size_t `nvnd`
The number of vendor models in this element

Health Server The Health Server model provides attention callbacks and node diagnostics for *Health Client* models. It is primarily used to report faults in the mesh node and map the mesh nodes to their physical location.

Faults The Health Server model may report a list of faults that have occurred in the device's lifetime. Typically, the faults are events or conditions that may alter the behavior of the node, like power outages or faulty peripherals. Faults are split into warnings and errors. Warnings indicate conditions that are close to the limits of what the node is designed to withstand, but not necessarily damaging to the device. Errors indicate conditions that are outside of the node's design limits, and may have caused invalid behavior or permanent damage to the device.

Fault values `0x01` to `0x7f` are reserved for the Bluetooth mesh specification, and the full list of specification defined faults are available in *Health faults*. Fault values `0x80` to `0xff` are vendor specific. The list of faults are always reported with a company ID to help interpreting the vendor specific faults.

Attention state The attention state is used to make the device call attention to itself through some physical behavior like blinking, playing a sound or vibrating. The attention state may be used during provisioning to let the user know which device they're provisioning, as well as through the Health models at runtime.

The attention state is always assigned a timeout in the range of one to 255 seconds when enabled. The Health Server API provides two callbacks for the application to run their attention calling behavior: *bt_mesh_health_srv_cb.attn_on* is called at the beginning of the attention period, *bt_mesh_health_srv_cb.attn_off* is called at the end.

The remaining time for the attention period may be queried through *bt_mesh_health_srv.attn_timer*.

API reference

group `bt_mesh_health_srv`

Health Server Model.

Defines

`BT_MESH_HEALTH_PUB_DEFINE(_name, _max_faults)`

A helper to define a health publication context

Parameters

- `_name` – Name given to the publication context variable.
- `_max_faults` – Maximum number of faults the element can have.

`BT_MESH_MODEL_HEALTH_SRV(srv, pub)`

Define a new health server model. Note that this API needs to be repeated for each element that the application wants to have a health server model on. Each instance also needs a unique *bt_mesh_health_srv* and *bt_mesh_model_pub* context.

Parameters

- `srv` – Pointer to a unique struct *bt_mesh_health_srv*.
- `pub` – Pointer to a unique struct *bt_mesh_model_pub*.

Returns New mesh model instance.

Functions

int `bt_mesh_fault_update`(struct *bt_mesh_elem* *elem)

Notify the stack that the fault array state of the given element has changed.

This prompts the Health server on this element to publish the current fault array if periodic publishing is disabled.

Parameters

- `elem` – Element to update the fault state of.

Returns 0 on success, or (negative) error code otherwise.

struct `bt_mesh_health_srv_cb`

#include <health_srv.h> Callback function for the Health Server model

Public Members

int (*`fault_get_cur`)(struct *bt_mesh_model* *model, uint8_t *test_id, uint16_t *company_id, uint8_t *faults, uint8_t *fault_count)

Callback for fetching current faults.

Fault values may either be defined by the specification, or by a vendor. Vendor specific faults should be interpreted in the context of the accompanying Company ID. Specification defined faults may be reported for any Company ID, and the same fault may be presented for multiple Company IDs.

All faults shall be associated with at least one Company ID, representing the device vendor or some other vendor whose vendor specific fault values are used.

If there are multiple Company IDs that have active faults, return only the faults associated with one of them at the time. To report faults for multiple Company IDs, interleave which Company ID is reported for each call.

Param model Health Server model instance to get faults of.
Param test_id Test ID response buffer.
Param company_id Company ID response buffer.
Param faults Array to fill with current faults.
Param fault_count The number of faults the fault array can fit. Should be updated to reflect the number of faults copied into the array.
Return 0 on success, or (negative) error code otherwise.

int (*`fault_get_reg`)(struct *bt_mesh_model* *model, uint16_t company_id, uint8_t *test_id, uint8_t *faults, uint8_t *fault_count)

Callback for fetching all registered faults.

Registered faults are all past and current faults since the last call to `fault_clear`. Only faults associated with the given Company ID should be reported.

Fault values may either be defined by the specification, or by a vendor. Vendor specific faults should be interpreted in the context of the accompanying Company ID. Specification defined faults may be reported for any Company ID, and the same fault may be presented for multiple Company IDs.

Param model Health Server model instance to get faults of.
Param company_id Company ID to get faults for.
Param test_id Test ID response buffer.
Param faults Array to fill with registered faults.
Param fault_count The number of faults the fault array can fit. Should be updated to reflect the number of faults copied into the array.
Return 0 on success, or (negative) error code otherwise.

int (*`fault_clear`)(struct *bt_mesh_model* *model, uint16_t company_id)

Clear all registered faults associated with the given Company ID.

Param model Health Server model instance to clear faults of.
Param company_id Company ID to clear faults for.
Return 0 on success, or (negative) error code otherwise.

int (*`fault_test`)(struct *bt_mesh_model* *model, uint8_t test_id, uint16_t company_id)

Run a self-test.

The Health server may support up to 256 self-tests for each Company ID. The behavior for all test IDs are vendor specific, and should be interpreted based on the accompanying Company ID. Test failures should result in changes to the fault array.

Param model Health Server model instance to run test for.
Param test_id Test ID to run.
Param company_id Company ID to run test for.
Return 0 if the test execution was started successfully, or (negative) error code otherwise. Note that the fault array will not be reported back to the client if the test execution didn't start.

void (*`attn_on`)(struct *bt_mesh_model* *model)

Start calling attention to the device.

The attention state is used to map an element address to a physical device. When this callback is called, the device should start some physical procedure meant to call attention to itself, like blinking, buzzing, vibrating or moving. If there are multiple Health server instances on the device, the attention state should also help identify the specific element the server is in.

The attention calling behavior should continue until the `attn_off` callback is called.

Param model Health Server model to start the attention state of.

void (*attn_off)(struct *bt_mesh_model* *model)

Stop the attention state.

Any physical activity started to call attention to the device should be stopped.

Param model

struct bt_mesh_health_srv

#include <health_srv.h> Mesh Health Server Model Context

Public Members

struct *bt_mesh_model* *model

Composition data model entry pointer.

const struct *bt_mesh_health_srv_cb* *cb

Optional callback struct

struct *k_work_delayable* attn_timer

Attention Timer state

Health faults

Fault values defined by the Bluetooth mesh specification.

group bt_mesh_health_faults

List of specification defined Health fault values.

Defines

BT_MESH_HEALTH_FAULT_NO_FAULT

No fault has occurred.

BT_MESH_HEALTH_FAULT_BATTERY_LOW_WARNING

BT_MESH_HEALTH_FAULT_BATTERY_LOW_ERROR

BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_LOW_WARNING

BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_LOW_ERROR

BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_HIGH_WARNING

BT_MESH_HEALTH_FAULT_SUPPLY_VOLTAGE_TOO_HIGH_ERROR

BT_MESH_HEALTH_FAULT_POWER_SUPPLY_INTERRUPTED_WARNING

BT_MESH_HEALTH_FAULT_POWER_SUPPLY_INTERRUPTED_ERROR

BT_MESH_HEALTH_FAULT_NO_LOAD_WARNING

BT_MESH_HEALTH_FAULT_NO_LOAD_ERROR

BT_MESH_HEALTH_FAULT_OVERLOAD_WARNING

BT_MESH_HEALTH_FAULT_OVERLOAD_ERROR

BT_MESH_HEALTH_FAULT_OVERHEAT_WARNING

BT_MESH_HEALTH_FAULT_OVERHEAT_ERROR

BT_MESH_HEALTH_FAULT_CONDENSATION_WARNING

BT_MESH_HEALTH_FAULT_CONDENSATION_ERROR

BT_MESH_HEALTH_FAULT_VIBRATION_WARNING

BT_MESH_HEALTH_FAULT_VIBRATION_ERROR

BT_MESH_HEALTH_FAULT_CONFIGURATION_WARNING

BT_MESH_HEALTH_FAULT_CONFIGURATION_ERROR

BT_MESH_HEALTH_FAULT_ELEMENT_NOT_CALIBRATED_WARNING

BT_MESH_HEALTH_FAULT_ELEMENT_NOT_CALIBRATED_ERROR

BT_MESH_HEALTH_FAULT_MEMORY_WARNING

BT_MESH_HEALTH_FAULT_MEMORY_ERROR

BT_MESH_HEALTH_FAULT_SELF_TEST_WARNING

BT_MESH_HEALTH_FAULT_SELF_TEST_ERROR

BT_MESH_HEALTH_FAULT_INPUT_TOO_LOW_WARNING

BT_MESH_HEALTH_FAULT_INPUT_TOO_LOW_ERROR

BT_MESH_HEALTH_FAULT_INPUT_TOO_HIGH_WARNING

BT_MESH_HEALTH_FAULT_INPUT_TOO_HIGH_ERROR

BT_MESH_HEALTH_FAULT_INPUT_NO_CHANGE_WARNING

BT_MESH_HEALTH_FAULT_INPUT_NO_CHANGE_ERROR

BT_MESH_HEALTH_FAULT_ACTUATOR_BLOCKED_WARNING

BT_MESH_HEALTH_FAULT_ACTUATOR_BLOCKED_ERROR

BT_MESH_HEALTH_FAULT_HOUSING_OPENED_WARNING

BT_MESH_HEALTH_FAULT_HOUSING_OPENED_ERROR

BT_MESH_HEALTH_FAULT_TAMPER_WARNING

BT_MESH_HEALTH_FAULT_TAMPER_ERROR

BT_MESH_HEALTH_FAULT_DEVICE_MOVED_WARNING

BT_MESH_HEALTH_FAULT_DEVICE_MOVED_ERROR

BT_MESH_HEALTH_FAULT_DEVICE_DROPPED_WARNING

BT_MESH_HEALTH_FAULT_DEVICE_DROPPED_ERROR

BT_MESH_HEALTH_FAULT_OVERFLOW_WARNING

BT_MESH_HEALTH_FAULT_OVERFLOW_ERROR

BT_MESH_HEALTH_FAULT_EMPTY_WARNING

BT_MESH_HEALTH_FAULT_EMPTY_ERROR

BT_MESH_HEALTH_FAULT_INTERNAL_BUS_WARNING

BT_MESH_HEALTH_FAULT_INTERNAL_BUS_ERROR

BT_MESH_HEALTH_FAULT_MECHANISM_JAMMED_WARNING

BT_MESH_HEALTH_FAULT_MECHANISM_JAMMED_ERROR

BT_MESH_HEALTH_FAULT_VENDOR_SPECIFIC_START

Start of the vendor specific fault values.

All values below this are reserved for the Bluetooth Specification.

Health Client The Health Client model interacts with a Health Server model to read out diagnostics and control the node's attention state.

All message passing functions in the Health Client API have `net_idx` and `addr` as their first parameters. These should be set to the network index and primary unicast address that the target node was provisioned with.

The Health Client model is optional, and may be instantiated in any element. However, if a Health Client model is instantiated in an element other than the first, an instance must also be present in the first element.

See *Health faults* for a list of specification defined fault values.

API reference

group `bt_mesh_health_cli`

Health Client Model.

Defines

`BT_MESH_MODEL_HEALTH_CLI(cli_data)`

Generic Health Client model composition data entry.

Parameters

- `cli_data` – Pointer to a *Health Client Model* instance.

Functions

int `bt_mesh_health_cli_set`(struct *bt_mesh_model* *model)

Set Health client model instance to use for communication.

Parameters

- `model` – Health Client model instance from the composition data.

Returns 0 on success, or (negative) error code on failure.

int `bt_mesh_health_fault_get`(uint16_t addr, uint16_t app_idx, uint16_t cid, uint8_t *test_id, uint8_t *faults, size_t *fault_count)

Get the registered fault state for the given Company ID.

See also:

Health faults

Parameters

- `addr` – Target node element address.
- `app_idx` – Application index to encrypt with.
- `cid` – Company ID to get the registered faults of.
- `test_id` – Test ID response buffer.
- `faults` – Fault array response buffer.
- `fault_count` – Fault count response buffer.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_health_fault_clear(uint16_t addr, uint16_t app_idx, uint16_t cid, uint8_t *test_id, uint8_t *faults, size_t *fault_count)

Clear the registered faults for the given Company ID.

See also:

Health faults

Parameters

- addr – Target node element address.
- app_idx – Application index to encrypt with.
- cid – Company ID to clear the registered faults for.
- test_id – Test ID response buffer.
- faults – Fault array response buffer.
- fault_count – Fault count response buffer.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_health_fault_test(uint16_t addr, uint16_t app_idx, uint16_t cid, uint8_t test_id, uint8_t *faults, size_t *fault_count)

Invoke a self-test procedure for the given Company ID.

Parameters

- addr – Target node element address.
- app_idx – Application index to encrypt with.
- cid – Company ID to invoke the test for.
- test_id – Test ID response buffer.
- faults – Fault array response buffer.
- fault_count – Fault count response buffer.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_health_period_get(uint16_t addr, uint16_t app_idx, uint8_t *divisor)

Get the target node's Health fast period divisor.

The health period divisor is used to increase the publish rate when a fault is registered. Normally, the Health server will publish with the period in the configured publish parameters. When a fault is registered, the publish period is divided by (1 << divisor). For example, if the target node's Health server is configured to publish with a period of 16 seconds, and the Health fast period divisor is 5, the Health server will publish with an interval of 500 ms when a fault is registered.

Parameters

- addr – Target node element address.
- app_idx – Application index to encrypt with.
- divisor – Health period divisor response buffer.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_health_period_set(uint16_t addr, uint16_t app_idx, uint8_t divisor, uint8_t *updated_divisor)

Set the target node's Health fast period divisor.

The health period divisor is used to increase the publish rate when a fault is registered. Normally, the Health server will publish with the period in the configured publish parameters. When a fault is registered, the publish period is divided by (1 << divisor). For example, if the target node's Health server is configured to publish with a period of 16 seconds, and the Health fast period divisor is 5, the Health server will publish with an interval of 500 ms when a fault is registered.

Parameters

- addr – Target node element address.
- app_idx – Application index to encrypt with.
- divisor – New Health period divisor.
- updated_divisor – Health period divisor response buffer.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_health_attention_get(uint16_t addr, uint16_t app_idx, uint8_t *attention)

Get the current attention timer value.

Parameters

- addr – Target node element address.
- app_idx – Application index to encrypt with.
- attention – Attention timer response buffer, measured in seconds.

Returns 0 on success, or (negative) error code on failure.

int bt_mesh_health_attention_set(uint16_t addr, uint16_t app_idx, uint8_t attention, uint8_t *updated_attention)

Set the attention timer.

Parameters

- addr – Target node element address.
- app_idx – Application index to encrypt with.
- attention – New attention timer time, in seconds.
- updated_attention – Attention timer response buffer, measured in seconds.

Returns 0 on success, or (negative) error code on failure.

int32_t bt_mesh_health_cli_timeout_get(void)

Get the current transmission timeout value.

Returns The configured transmission timeout in milliseconds.

void bt_mesh_health_cli_timeout_set(int32_t timeout)

Set the transmission timeout value.

Parameters

- timeout – The new transmission timeout.

struct bt_mesh_health_cli

#include <health_cli.h> Health Client Model Context

Public Members

struct *bt_mesh_model* *model

Composition data model entry pointer.

void (*current_status)(struct *bt_mesh_health_cli* *cli, uint16_t addr, uint8_t test_id, uint16_t cid, uint8_t *faults, size_t fault_count)

Optional callback for Health Current Status messages.

Handles received Health Current Status messages from a Health server. The fault array represents all faults that are currently present in the server's element.

See also:

Health faults

Param cli Health client that received the status message.
Param addr Address of the sender.
Param test_id Identifier of a most recently performed test.
Param cid Company Identifier of the node.
Param faults Array of faults.
Param fault_count Number of faults in the fault array.

Message

The Bluetooth mesh message provides set of structures, macros and functions used for preparing message buffers, managing message and acknowledged message contexts.

API reference

group bt_mesh_msg

Message.

Defines

BT_MESH_MIC_SHORT

Length of a short Mesh MIC.

BT_MESH_MIC_LONG

Length of a long Mesh MIC.

BT_MESH_MODEL_OP_LEN(_op)

Helper to determine the length of an opcode.

Parameters

- _op – Opcode.

BT_MESH_MODEL_BUF_LEN(_op, _payload_len)

Helper for model message buffer length.

Returns the length of a Mesh model message buffer, including the opcode length and a short MIC.

Parameters

- `_op` – Opcode of the message.
- `_payload_len` – Length of the model payload.

`BT_MESH_MODEL_BUF_LEN_LONG_MIC(_op, _payload_len)`

Helper for model message buffer length.

Returns the length of a Mesh model message buffer, including the opcode length and a long MIC.

Parameters

- `_op` – Opcode of the message.
- `_payload_len` – Length of the model payload.

`BT_MESH_MODEL_BUF_DEFINE(_buf, _op, _payload_len)`

Define a Mesh model message buffer using *NET_BUF_SIMPLE_DEFINE*.

Parameters

- `_buf` – Buffer name.
- `_op` – Opcode of the message.
- `_payload_len` – Length of the model message payload.

Functions

void `bt_mesh_model_msg_init`(struct *net_buf_simple* *msg, uint32_t opcode)

Initialize a model message.

Clears the message buffer contents, and encodes the given opcode. The message buffer will be ready for filling in payload data.

Parameters

- `msg` – Message buffer.
- `opcode` – Opcode to encode.

static inline void `bt_mesh_msg_ack_ctx_init`(struct *bt_mesh_msg_ack_ctx* *ack)

Initialize an acknowledged message context.

Initializes semaphore used for synchronization between *bt_mesh_msg_ack_ctx_wait* and *bt_mesh_msg_ack_ctx_rx* calls. Call this function before using *bt_mesh_msg_ack_ctx*.

Parameters

- `ack` – Acknowledged message context to initialize.

static inline void `bt_mesh_msg_ack_ctx_reset`(struct *bt_mesh_msg_ack_ctx* *ack)

Reset the synchronization semaphore in an acknowledged message context.

This function aborts call to *bt_mesh_msg_ack_ctx_wait*.

Parameters

- `ack` – Acknowledged message context to be reset.

void `bt_mesh_msg_ack_ctx_clear`(struct *bt_mesh_msg_ack_ctx* *ack)

Clear parameters of an acknowledged message context.

This function clears the opcode, remote address and user data set by *bt_mesh_msg_ack_ctx_prepare*.

Parameters

- `ack` – Acknowledged message context to be cleared.

int bt_mesh_msg_ack_ctx_prepare(struct *bt_mesh_msg_ack_ctx* *ack, uint32_t op, uint16_t dst, void *user_data)

Prepare an acknowledged message context for the incoming message to wait.

This function sets the opcode, remote address of the incoming message and stores the user data. Use this function before calling *bt_mesh_msg_ack_ctx_wait*.

Parameters

- ack – Acknowledged message context to prepare.
- op – The message OpCode.
- dst – Destination address of the message.
- user_data – User data for the acknowledged message context.

Returns 0 on success, or (negative) error code on failure.

static inline bool bt_mesh_msg_ack_ctx_busy(struct *bt_mesh_msg_ack_ctx* *ack)

Check if the acknowledged message context is initialized with an opcode.

Parameters

- ack – Acknowledged message context.

Returns true if the acknowledged message context is initialized with an opcode, false otherwise.

int bt_mesh_msg_ack_ctx_wait(struct *bt_mesh_msg_ack_ctx* *ack, *k_timeout_t* timeout)

Wait for a message acknowledge.

This function blocks execution until *bt_mesh_msg_ack_ctx_rx* is called or by timeout.

Parameters

- ack – Acknowledged message context of the message to wait for.
- timeout – Wait timeout.

Returns 0 on success, or (negative) error code on failure.

static inline void bt_mesh_msg_ack_ctx_rx(struct *bt_mesh_msg_ack_ctx* *ack)

Mark a message as acknowledged.

This function unblocks call to *bt_mesh_msg_ack_ctx_wait*.

Parameters

- ack – Context of a message to be acknowledged.

bool bt_mesh_msg_ack_ctx_match(const struct *bt_mesh_msg_ack_ctx* *ack, uint32_t op, uint16_t addr, void **user_data)

Check if an opcode and address of a message matches the expected one.

Parameters

- ack – Acknowledged message context to be checked.
- op – OpCode of the incoming message.
- addr – Source address of the incoming message.
- user_data – If not NULL, returns a user data stored in the acknowledged message context by *bt_mesh_msg_ack_ctx_prepare*.

Returns true if the incoming message matches the expected one, false otherwise.

struct bt_mesh_msg_ctx

#include <msg.h> Message sending context.

Public Members

uint16_t `net_idx`

NetKey Index of the subnet to send the message on.

uint16_t `app_idx`

AppKey Index to encrypt the message with.

uint16_t `addr`

Remote address.

uint16_t `recv_dst`

Destination address of a received message. Not used for sending.

int8_t `recv_rssi`

RSSI of received packet. Not used for sending.

uint8_t `recv_ttl`

Received TTL value. Not used for sending.

bool `send_rel`

Force sending reliably by using segment acknowledgment

uint8_t `send_ttl`

TTL, or BT_MESH_TTL_DEFAULT for default TTL.

struct `bt_mesh_msg_ack_ctx`

#include <msg.h> Acknowledged message context for tracking the status of model messages pending a response.

Public Members

struct k_sem `sem`

Sync semaphore.

uint32_t `op`

Opcode we're waiting for.

uint16_t `dst`

Address of the node that should respond.

void *`user_data`

User specific parameter.

Provisioning

Provisioning is the process of adding devices to a mesh network. It requires two devices operating in the following roles:

- The *provisioner* represents the network owner, and is responsible for adding new nodes to the mesh network.
- The *provisionee* is the device that gets added to the network through the Provisioning process. Before the provisioning process starts, the provisionee is an *unprovisioned device*.

The Provisioning module in the Zephyr Bluetooth mesh stack supports both the Advertising and GATT Provisioning bearers for the provisionee role, as well as the Advertising Provisioning bearer for the provisioner role.

The Provisioning process All Bluetooth mesh nodes must be provisioned before they can participate in a Bluetooth mesh network. The Provisioning API provides all the functionality necessary for a device to become a provisioned mesh node. Provisioning is a five-step process, involving the following steps:

- Beaconing
- Invitation
- Public key exchange
- Authentication
- Provisioning data transfer

Beaconing To start the provisioning process, the unprovisioned device must first start broadcasting the Unprovisioned Beacon. This makes it visible to nearby provisioners, which can initiate the provisioning. To indicate that the device needs to be provisioned, call *bt_mesh_prov_enable()*. The device starts broadcasting the Unprovisioned Beacon with the device UUID and the `OOB information` field, as specified in the `prov` parameter passed to *bt_mesh_init()*. Additionally, a Uniform Resource Identifier (URI) may be specified, which can point the provisioner to the location of some Out Of Band information, such as the device's public key or an authentication value database. The URI is advertised in a separate beacon, with a URI hash included in the unprovisioned beacon, to tie the two together.

Uniform Resource Identifier The Uniform Resource Identifier shall follow the format specified in the Bluetooth Core Specification Supplement. The URI must start with a URI scheme, encoded as a single utf-8 data point, or the special `none` scheme, encoded as `0x01`. The available schemes are listed on the Bluetooth website.

Examples of encoded URIs:

Table 2: URI encoding examples

URI	Encoded
`http://example.com`	`\x16//example.com`
`https://www.zephyrproject.org/`	`\x17//www.zephyrproject.org/`
`just a string`	`\x01just a string`

Provisioning invitation The provisioner initiates the Provisioning process by sending a Provisioning invitation. The invitations prompts the provisionee to call attention to itself using the Health Server *Attention state*, if available.

The Unprovisioned device automatically responds to the invite by presenting a list of its capabilities, including the supported Out of Band Authentication methods.

Public key exchange Before the provisioning process can begin, the provisioner and the unprovisioned device exchange public keys, either in-band or Out of Band (OOB).

In-band public key exchange is a part of the provisioning process and always supported by the unprovisioned device and provisioner.

If the application wants to support public key exchange via OOB, it needs to provide public and private keys to the mesh stack. The unprovisioned device will reflect this in its capabilities. The provisioner obtains the public key via any available OOB mechanism (e.g. the device may advertise a packet containing the public key or it can be encoded in a QR code printed on the device packaging). Note that even if the unprovisioned device has specified the public key for the Out of Band exchange, the provisioner may choose to exchange the public key in-band if it can't retrieve the public key via OOB mechanism. In this case, a new key pair will be generated by the mesh stack for each Provisioning process.

To enable support of OOB public key on the unprovisioned device side, `CONFIG_BT_MESH_PROV_OOB_PUBLIC_KEY` needs to be enabled. The application must provide public and private keys before the Provisioning process is started by initializing pointers to *bt_mesh_prov.public_key_be* and *bt_mesh_prov.private_key_be*. The keys needs to be provided in big-endian bytes order.

To provide the device's public key obtained via OOB, call *bt_mesh_prov_remote_pub_key_set()* on the provisioner side.

Authentication After the initial exchange, the provisioner selects an Out of Band (OOB) Authentication method. This allows the user to confirm that the device the provisioner connected to is actually the device they intended, and not a malicious third party.

The Provisioning API supports the following authentication methods for the provisionee:

- **Static OOB:** An authentication value is assigned to the device in production, which the provisioner can query in some application specific way.
- **Input OOB:** The user inputs the authentication value. The available input actions are listed in *bt_mesh_input_action_t*.
- **Output OOB:** Show the user the authentication value. The available output actions are listed in *bt_mesh_output_action_t*.

The application must provide callbacks for the supported authentication methods in *bt_mesh_prov*, as well as enabling the supported actions in *bt_mesh_prov.output_actions* and *bt_mesh_prov.input_actions*.

When an Output OOB action is selected, the authentication value should be presented to the user when the output callback is called, and remain until the *bt_mesh_prov.input_complete* or *bt_mesh_prov.complete* callback is called. If the action is `blink`, `beep` or `vibrate`, the sequence should be repeated after a delay of three seconds or more.

When an Input OOB action is selected, the user should be prompted when the application receives the *bt_mesh_prov.input* callback. The user response should be fed back to the Provisioning API through *bt_mesh_input_string()* or *bt_mesh_input_number()*. If no user response is recorded within 60 seconds, the Provisioning process is aborted.

Data transfer After the device has been successfully authenticated, the provisioner transfers the Provisioning data:

- Unicast address
- A network key
- IV index
- Network flags
  - Key refresh
  - IV update

Additionally, a device key is generated for the node. All this data is stored by the mesh stack, and the provisioning *bt_mesh_prov.complete* callback gets called.

Provisioning security Depending on the choice of public key exchange mechanism and authentication method, the provisioning process can be secure or insecure.

On May 24th 2021, ANSSI disclosed a set of vulnerabilities in the Bluetooth mesh provisioning protocol that showcased how the low entropy provided by the Blink, Vibrate, Push, Twist and Input/Output numeric OOB methods could be exploited in impersonation and MITM attacks. In response, the Bluetooth SIG has reclassified these OOB methods as insecure in the Mesh Profile specification erratum 16350, as AuthValue may be brute forced in real time. To ensure secure provisioning, applications should use a static OOB value and OOB public key transfer.

API reference

group `bt_mesh_prov`

Provisioning.

Enums

enum `bt_mesh_output_action_t`

Available Provisioning output authentication actions.

Values:

enumerator `BT_MESH_NO_OUTPUT` = 0

enumerator `BT_MESH_BLINK` = *BIT*(0)

enumerator `BT_MESH_BEEP` = *BIT*(1)

enumerator `BT_MESH_VIBRATE` = *BIT*(2)

enumerator `BT_MESH_DISPLAY_NUMBER` = *BIT*(3)

enumerator `BT_MESH_DISPLAY_STRING` = *BIT*(4)

enum `bt_mesh_input_action_t`

Available Provisioning input authentication actions.

Values:

enumerator `BT_MESH_NO_INPUT` = 0

enumerator `BT_MESH_PUSH` = *BIT*(0)

enumerator `BT_MESH_TWIST` = *BIT*(1)

enumerator `BT_MESH_ENTER_NUMBER` = *BIT*(2)

enumerator `BT_MESH_ENTER_STRING` = *BIT*(3)

enum bt_mesh_prov_bearer_t

Available Provisioning bearers.

Values:

enumerator BT_MESH_PROV_ADV = *BIT*(0)

enumerator BT_MESH_PROV_GATT = *BIT*(1)

enum bt_mesh_prov_oob_info_t

Out of Band information location.

Values:

enumerator BT_MESH_PROV_OOB_OTHER = *BIT*(0)

enumerator BT_MESH_PROV_OOB_URI = *BIT*(1)

enumerator BT_MESH_PROV_OOB_2D_CODE = *BIT*(2)

enumerator BT_MESH_PROV_OOB_BAR_CODE = *BIT*(3)

enumerator BT_MESH_PROV_OOB_NFC = *BIT*(4)

enumerator BT_MESH_PROV_OOB_NUMBER = *BIT*(5)

enumerator BT_MESH_PROV_OOB_STRING = *BIT*(6)

enumerator BT_MESH_PROV_OOB_ON_BOX = *BIT*(11)

enumerator BT_MESH_PROV_OOB_IN_BOX = *BIT*(12)

enumerator BT_MESH_PROV_OOB_ON_PAPER = *BIT*(13)

enumerator BT_MESH_PROV_OOB_IN_MANUAL = *BIT*(14)

enumerator BT_MESH_PROV_OOB_ON_DEV = *BIT*(15)

Functions

int bt_mesh_input_string(const char *str)

Provide provisioning input OOB string.

This is intended to be called after the *bt_mesh_prov* input callback has been called with BT_MESH_ENTER_STRING as the action.

Parameters

- str – String.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_input_number`(uint32_t num)

Provide provisioning input OOB number.

This is intended to be called after the *bt_mesh_prov* input callback has been called with BT_MESH_ENTER_NUMBER as the action.

Parameters

- `num` – Number.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_prov_remote_pub_key_set`(const uint8_t public_key[64])

Provide Device public key.

Parameters

- `public_key` – Device public key in big-endian.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_auth_method_set_input`(*bt_mesh_input_action_t* action, uint8_t size)

Use Input OOB authentication.

Provisioner only.

Instruct the unprovisioned device to use the specified Input OOB authentication action. When using *BT_MESH_PUSH*, *BT_MESH_TWIST* or *BT_MESH_ENTER_NUMBER*, the *bt_mesh_prov::output_number* callback is called with a random number that has to be entered on the unprovisioned device.

When using *BT_MESH_ENTER_STRING*, the *bt_mesh_prov::output_string* callback is called with a random string that has to be entered on the unprovisioned device.

Parameters

- `action` – Authentication action used by the unprovisioned device.
- `size` – Authentication size.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_auth_method_set_output`(*bt_mesh_output_action_t* action, uint8_t size)

Use Output OOB authentication.

Provisioner only.

Instruct the unprovisioned device to use the specified Output OOB authentication action. The *bt_mesh_prov::input* callback will be called.

When using *BT_MESH_BLINK*, *BT_MESH_BEEP*, *BT_MESH_VIBRATE* or *BT_MESH_DISPLAY_NUMBER*, and the application has to call *bt_mesh_input_number* with the random number indicated by the unprovisioned device.

When using *BT_MESH_DISPLAY_STRING*, the application has to call *bt_mesh_input_string* with the random string displayed by the unprovisioned device.

Parameters

- `action` – Authentication action used by the unprovisioned device.
- `size` – Authentication size.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_auth_method_set_static`(const uint8_t *static_val, uint8_t size)

Use static OOB authentication.

Provisioner only.

Instruct the unprovisioned device to use static OOB authentication, and use the given static authentication value when provisioning.

Parameters

- `static_val` – Static OOB value.
- `size` – Static OOB value size.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_auth_method_set_none`(void)

Don't use OOB authentication.

Provisioner only.

Don't use any authentication when provisioning new devices. This is the default behavior.

Warning: Not using any authentication exposes the mesh network to impersonation attacks, where attackers can pretend to be the unprovisioned device to gain access to the network. Authentication is strongly encouraged.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_prov_enable`(*bt_mesh_prov_bearer_t* bearers)

Enable specific provisioning bearers.

Enable one or more provisioning bearers.

Parameters

- `bearers` – Bit-wise or of provisioning bearers.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_prov_disable`(*bt_mesh_prov_bearer_t* bearers)

Disable specific provisioning bearers.

Disable one or more provisioning bearers.

Parameters

- `bearers` – Bit-wise or of provisioning bearers.

Returns Zero on success or (negative) error code otherwise.

int `bt_mesh_provision`(const uint8_t net_key[16], uint16_t net_idx, uint8_t flags, uint32_t iv_index, uint16_t addr, const uint8_t dev_key[16])

Provision the local Mesh Node.

This API should normally not be used directly by the application. The only exception is for testing purposes where manual provisioning is desired without an actual external provisioner.

Parameters

- `net_key` – Network Key
- `net_idx` – Network Key Index
- `flags` – Provisioning Flags
- `iv_index` – IV Index
- `addr` – Primary element address
- `dev_key` – Device Key

Returns Zero on success or (negative) error code otherwise.

int bt_mesh_provision_adv(const uint8_t uuid[16], uint16_t net_idx, uint16_t addr, uint8_t attention_duration)

Provision a Mesh Node using PB-ADV.

Parameters

- uuid – UUID
- net_idx – Network Key Index
- addr – Address to assign to remote device. If addr is 0, the lowest available address will be chosen.
- attention_duration – The attention duration to be send to remote device

Returns Zero on success or (negative) error code otherwise.

bool bt_mesh_is_provisioned(void)

Check if the local node has been provisioned.

This API can be used to check if the local node has been provisioned or not. It can e.g. be helpful to determine if there was a stored network in flash, i.e. if the network was restored after calling *settings_load()*.

Returns True if the node is provisioned. False otherwise.

struct bt_mesh_dev_capabilities

#include <main.h> Device Capabilities.

Public Members

uint8_t elem_count

Number of elements supported by the device

uint16_t algorithms

Supported algorithms and other capabilities

uint8_t pub_key_type

Supported public key types

uint8_t static_oob

Supported static OOB Types

bt_mesh_output_action_t output_actions

Supported Output OOB Actions

bt_mesh_input_action_t input_actions

Supported Input OOB Actions

uint8_t output_size

Maximum size of Output OOB supported

uint8_t input_size

Maximum size in octets of Input OOB supported

struct `bt_mesh_prov`

#include <main.h> Provisioning properties & capabilities.

Public Members

const uint8_t *`uuid`

The UUID that's used when advertising as unprovisioned

const char *`uri`

Optional URI. This will be advertised separately from the unprovisioned beacon, however the unprovisioned beacon will contain a hash of it so the two can be associated by the provisioner.

bt_mesh_prov_oob_info_t `oob_info`

Out of Band information field.

const uint8_t *`public_key_be`

Pointer to Public Key in big-endian for OOB public key type support.

Remember to enable `CONFIG_BT_MESH_PROV_OOB_PUBLIC_KEY` when initializing this parameter.

Must be used together with *bt_mesh_prov::private_key_be*.

const uint8_t *`private_key_be`

Pointer to Private Key in big-endian for OOB public key type support.

Remember to enable `CONFIG_BT_MESH_PROV_OOB_PUBLIC_KEY` when initializing this parameter.

Must be used together with *bt_mesh_prov::public_key_be*.

const uint8_t *`static_val`

Static OOB value

uint8_t `static_val_len`

Static OOB value length

uint8_t `output_size`

Maximum size of Output OOB supported

uint16_t `output_actions`

Supported Output OOB Actions

uint8_t `input_size`

Maximum size of Input OOB supported

uint16_t `input_actions`

Supported Input OOB Actions

void (*capabilities)(const struct *bt_mesh_dev_capabilities* *cap)

Provisioning Capabilities.

This callback notifies the application that the provisioning capabilities of the unprovisioned device has been received.

The application can consequently call bt_mesh_auth_method_set_<*> to select suitable provisioning oob authentication method.

When this callback returns, the provisioner will start authentication with the chosen method.

Param cap capabilities supported by device.

int (*output_number)(*bt_mesh_output_action_t* act, uint32_t num)

Output of a number is requested.

This callback notifies the application that it should output the given number using the given action.

Param act Action for outputting the number.
Param num Number to be outputted.
Return Zero on success or negative error code otherwise

int (*output_string)(const char *str)

Output of a string is requested.

This callback notifies the application that it should display the given string to the user.

Param str String to be displayed.
Return Zero on success or negative error code otherwise

int (*input)(*bt_mesh_input_action_t* act, uint8_t size)

Input is requested.

This callback notifies the application that it should request input from the user using the given action. The requested input will either be a string or a number, and the application needs to consequently call the *bt_mesh_input_string()* or *bt_mesh_input_number()* functions once the data has been acquired from the user.

Param act Action for inputting data.
Param num Maximum size of the inputted data.
Return Zero on success or negative error code otherwise

void (*input_complete)(void)

The other device finished their OOB input.

This callback notifies the application that it should stop displaying its output OOB value, as the other party finished their OOB input.

void (*unprovisioned_beacon)(uint8_t uuid[16], *bt_mesh_prov_oob_info_t* oob_info, uint32_t *uri_hash)

Unprovisioned beacon has been received.

This callback notifies the application that an unprovisioned beacon has been received.

Param uuid UUID
Param oob_info OOB Information
Param uri_hash Pointer to URI Hash value. NULL if no hash was present in the beacon.

void (*link_open)(*bt_mesh_prov_bearer_t* bearer)

Provisioning link has been opened.

This callback notifies the application that a provisioning link has been opened on the given provisioning bearer.

Param bearer Provisioning bearer.

void (*`link_close`)(*bt_mesh_prov_bearer_t* bearer)

Provisioning link has been closed.

This callback notifies the application that a provisioning link has been closed on the given provisioning bearer.

Param bearer Provisioning bearer.

void (*`complete`)(uint16_t net_idx, uint16_t addr)

Provisioning is complete.

This callback notifies the application that provisioning has been successfully completed, and that the local node has been assigned the specified NetKeyIndex and primary element address.

Param net_idx NetKeyIndex given during provisioning.
Param addr Primary element address.

void (*`node_added`)(uint16_t net_idx, uint8_t uuid[16], uint16_t addr, uint8_t num_elem)

A new node has been added to the provisioning database.

This callback notifies the application that provisioning has been successfully completed, and that a node has been assigned the specified NetKeyIndex and primary element address.

Param net_idx NetKeyIndex given during provisioning.
Param uuid UUID of the added node
Param addr Primary element address.
Param num_elem Number of elements that this node has.

void (*`reset`)(void)

Node has been reset.

This callback notifies the application that the local node has been reset and needs to be reprovisioned. The node will not automatically advertise as unprovisioned, rather the *bt_mesh_prov_enable()* API needs to be called to enable unprovisioned advertising on one or more provisioning bearers.

Proxy

The Proxy feature allows legacy devices like phones to access the Bluetooth mesh network through GATT. The Proxy feature is only compiled in if the `CONFIG_BT_MESH_GATT_PROXY` option is set. The Proxy feature state is controlled by the *Configuration Server*, and the initial value can be set with `bt_mesh_cfg_srv.gatt_proxy`.

API reference

group `bt_mesh_proxy`

Proxy.

Defines

BT_MESH_PROXY_CB_DEFINE(_name)

Register a callback structure for Proxy events.

Registers a structure with callback functions that gets called on various Proxy events.

Parameters

- _name – Name of callback structure.

Functions

int bt_mesh_proxy_identity_enable(void)

Enable advertising with Node Identity.

This API requires that GATT Proxy support has been enabled. Once called each subnet will start advertising using Node Identity for the next 60 seconds.

Returns 0 on success, or (negative) error code on failure.

struct bt_mesh_proxy_cb

#include <proxy.h> Callbacks for the Proxy feature.

Should be instantiated with *BT_MESH_PROXY_CB_DEFINE*.

Public Members

void (*identity_enabled)(uint16_t net_idx)

Started sending Node Identity beacons on the given subnet.
Param net_idx Network index the Node Identity beacons are running on.

void (*identity_disabled)(uint16_t net_idx)

Stopped sending Node Identity beacons on the given subnet.
Param net_idx Network index the Node Identity beacons were running on.

Heartbeat

The Heartbeat feature provides functionality for monitoring Bluetooth mesh nodes and determining the distance between nodes.

The Heartbeat feature is configured through the *Configuration Server* model.

Heartbeat messages Heartbeat messages are sent as transport control packets through the network, and are only encrypted with a network key. Heartbeat messages contain the original Time To Live (TTL) value used to send the message and a bitfield of the active features on the node. Through this, a receiving node can determine how many relays the message had to go through to arrive at the receiver, and what features the node supports.

Available Heartbeat feature flags:

- *BT_MESH_FEAT_RELAY*
- *BT_MESH_FEAT_PROXY*
- *BT_MESH_FEAT_FRIEND*
- *BT_MESH_FEAT_LOW_POWER*

Heartbeat publication Heartbeat publication is controlled through the Configuration models, and can be triggered in two ways:

Periodic publication The node publishes a new Heartbeat message at regular intervals. The publication can be configured to stop after a certain number of messages, or continue indefinitely.

Triggered publication The node publishes a new Heartbeat message every time a feature changes. The set of features that can trigger the publication is configurable.

The two publication types can be combined.

Heartbeat subscription A node can be configured to subscribe to Heartbeat messages from one node at the time. To receive a Heartbeat message, both the source and destination must match the configured subscription parameters.

Heartbeat subscription is always time limited, and throughout the subscription period, the node keeps track of the number of received Heartbeats as well as the minimum and maximum received hop count.

All Heartbeats received with the configured subscription parameters are passed to the `bt_mesh_hb_cb::recv` event handler.

When the Heartbeat subscription period ends, the `bt_mesh_hb_cb::sub_end` callback gets called.

API reference

group `bt_mesh_heartbeat`

Heartbeat.

Defines

`BT_MESH_HB_CB_DEFINE(_name)`

Register a callback structure for Heartbeat events.

Registers a callback structure that will be called whenever Heartbeat events occur

Parameters

- `_name` – Name of callback structure.

Functions

void `bt_mesh_hb_pub_get`(struct *bt_mesh_hb_pub* *get)

Get the current Heartbeat publication parameters.

Parameters

- `get` – Heartbeat publication parameters return buffer.

void `bt_mesh_hb_sub_get`(struct *bt_mesh_hb_sub* *get)

Get the current Heartbeat subscription parameters.

Parameters

- `get` – Heartbeat subscription parameters return buffer.

struct `bt_mesh_hb_pub`

#include <heartbeat.h> Heartbeat Publication parameters

Public Members

uint16_t `dst`

Destination address.

uint16_t `count`

Remaining publish count.

uint8_t `ttl`

Time To Live value.

uint16_t `feat`

Bitmap of features that trigger a Heartbeat publication if they change. Legal values are *BT_MESH_FEAT_RELAY*, *BT_MESH_FEAT_PROXY*, *BT_MESH_FEAT_FRIEND* and *BT_MESH_FEAT_LOW_POWER*.

uint16_t `net_idx`

Network index used for publishing.

uint32_t `period`

Publication period in seconds.

struct `bt_mesh_hb_sub`

#include <heartbeat.h> Heartbeat Subscription parameters.

Public Members

uint32_t `period`

Subscription period in seconds.

uint32_t `remaining`

Remaining subscription time in seconds.

uint16_t `src`

Source address to receive Heartbeats from.

uint16_t `dst`

Destination address to received Heartbeats on.

uint16_t `count`

The number of received Heartbeat messages so far.

uint8_t `min_hops`

Minimum hops in received messages, ie the shortest registered path from the publishing node to the subscribing node. A Heartbeat received from an immediate neighbor has hop count = 1.

uint8_t `max_hops`

Maximum hops in received messages, ie the longest registered path from the publishing node to the subscribing node. A Heartbeat received from an immediate neighbor has hop count = 1.

struct `bt_mesh_hb_cb`

#include <heartbeat.h> Heartbeat callback structure

Public Members

void (*`recv`)(const struct *bt_mesh_hb_sub* *sub, uint8_t hops, uint16_t feat)

Receive callback for heartbeats.

Gets called on every received Heartbeat that matches the current Heartbeat subscription parameters.

Param sub Current Heartbeat subscription parameters.

Param hops The number of hops the Heartbeat was received with.

Param feat The feature set of the publishing node. The value is a bitmap of *BT_MESH_FEAT_RELAY*, *BT_MESH_FEAT_PROXY*, *BT_MESH_FEAT_FRIEND* and *BT_MESH_FEAT_LOW_POWER*.

void (*`sub_end`)(const struct *bt_mesh_hb_sub* *sub)

Subscription end callback for heartbeats.

Gets called when the subscription period ends, providing a summary of the received heartbeat messages.

Param sub Current Heartbeat subscription parameters.

Runtime Configuration

The runtime configuration API allows applications to change their runtime configuration directly, without going through the Configuration models.

Bluetooth mesh nodes should generally be configured by a central network configurator device with a *Configuration Client* model. Each mesh node instantiates a *Configuration Server* model that the Configuration Client can communicate with to change the node configuration. In some cases, the mesh node can't rely on the Configuration Client to detect or determine local constraints, such as low battery power or changes in topology. For these scenarios, this API can be used to change the configuration locally.

API reference

group `bt_mesh_cfg`

Runtime Configuration.

Defines

`BT_MESH_KR_NORMAL`

`BT_MESH_KR_PHASE_1`

BT_MESH_KR_PHASE_2

BT_MESH_KR_PHASE_3

BT_MESH_RELAY_DISABLED

BT_MESH_RELAY_ENABLED

BT_MESH_RELAY_NOT_SUPPORTED

BT_MESH_BEACON_DISABLED

BT_MESH_BEACON_ENABLED

BT_MESH_GATT_PROXY_DISABLED

BT_MESH_GATT_PROXY_ENABLED

BT_MESH_GATT_PROXY_NOT_SUPPORTED

BT_MESH_FRIEND_DISABLED

BT_MESH_FRIEND_ENABLED

BT_MESH_FRIEND_NOT_SUPPORTED

BT_MESH_NODE_IDENTITY_STOPPED

BT_MESH_NODE_IDENTITY_RUNNING

BT_MESH_NODE_IDENTITY_NOT_SUPPORTED

Enums

enum bt_mesh_feat_state

Bluetooth mesh feature states

Values:

enumerator BT_MESH_FEATURE_DISABLED

Feature is supported, but disabled.

enumerator BT_MESH_FEATURE_ENABLED

Feature is supported and enabled.

enumerator BT_MESH_FEATURE_NOT_SUPPORTED

Feature is not supported, and cannot be enabled.

Functions

void `bt_mesh_beacon_set`(bool beacon)

Enable or disable sending of the Secure Network Beacon.

Parameters

- `beacon` – New Secure Network Beacon state.

bool `bt_mesh_beacon_enabled`(void)

Get the current Secure Network Beacon state.

Returns Whether the Secure Network Beacon feature is enabled.

int `bt_mesh_default_ttl_set`(uint8_t default_ttl)

Set the default TTL value.

The default TTL value is used when no explicit TTL value is set. Models will use the default TTL value when *bt_mesh_msg_ctx::send_ttl* is *BT_MESH_TTL_DEFAULT*.

Parameters

- `default_ttl` – The new default TTL value. Valid values are 0x00 and 0x02 to *BT_MESH_TTL_MAX*.

Return values

- `0` – Successfully set the default TTL value.
- `-EINVAL` – Invalid TTL value.

uint8_t `bt_mesh_default_ttl_get`(void)

Get the current default TTL value.

Returns The current default TTL value.

void `bt_mesh_net_transmit_set`(uint8_t xmit)

Set the Network Transmit parameters.

The Network Transmit parameters determine the parameters local messages are transmitted with.

See also:

BT_MESH_TRANSMIT

Parameters

- `xmit` – New Network Transmit parameters. Use *BT_MESH_TRANSMIT* for encoding.

uint8_t `bt_mesh_net_transmit_get`(void)

Get the current Network Transmit parameters.

The *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT* macros can be used to decode the Network Transmit parameters.

Returns The current Network Transmit parameters.

int `bt_mesh_relay_set`(enum *bt_mesh_feat_state* relay, uint8_t xmit)

Configure the Relay feature.

Enable or disable the Relay feature, and configure the parameters to transmit relayed messages with.

Support for the Relay feature must be enabled through the CONFIG_BT_MESH_RELAY configuration option.

See also:

BT_MESH_TRANSMIT

Parameters

- relay – New Relay feature state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.
- xmit – New Relay retransmit parameters. Use *BT_MESH_TRANSMIT* for encoding.

Return values

- 0 – Successfully changed the Relay configuration.
- -ENOTSUP – The Relay feature is not supported.
- -EINVAL – Invalid parameter.
- -EALREADY – Already using the given parameters.

enum *bt_mesh_feat_state* bt_mesh_relay_get(void)

Get the current Relay feature state.

Returns The Relay feature state.

uint8_t bt_mesh_relay_retransmit_get(void)

Get the current Relay Retransmit parameters.

The *BT_MESH_TRANSMIT_COUNT* and *BT_MESH_TRANSMIT_INT* macros can be used to decode the Relay Retransmit parameters.

Returns The current Relay Retransmit parameters, or 0 if relay is not supported.

int bt_mesh_gatt_proxy_set(enum *bt_mesh_feat_state* gatt_proxy)

Enable or disable the GATT Proxy feature.

Support for the GATT Proxy feature must be enabled through the CONFIG_BT_MESH_GATT_PROXY configuration option.

Note: The GATT Proxy feature only controls a Proxy node's ability to relay messages to the mesh network. A node that supports GATT Proxy will still advertise Connectable Proxy beacons, even if the feature is disabled. The Proxy feature can only be fully disabled through compile time configuration.

Parameters

- gatt_proxy – New GATT Proxy state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.

Return values

- 0 – Successfully changed the GATT Proxy feature state.
- -ENOTSUP – The GATT Proxy feature is not supported.
- -EINVAL – Invalid parameter.
- -EALREADY – Already in the given state.

enum *bt_mesh_feat_state* `bt_mesh_gatt_proxy_get`(void)

Get the current GATT Proxy state.

Returns The GATT Proxy feature state.

int `bt_mesh_friend_set`(enum *bt_mesh_feat_state* friendship)

Enable or disable the Friend feature.

Any active friendships will be terminated immediately if the Friend feature is disabled.

Support for the Friend feature must be enabled through the `CONFIG_BT_MESH_FRIEND` configuration option.

Parameters

- `friendship` – New Friend feature state. Must be one of *BT_MESH_FEATURE_ENABLED* and *BT_MESH_FEATURE_DISABLED*.

Return values

- `0` – Successfully changed the Friend feature state.
- `-ENOTSUP` – The Friend feature is not supported.
- `-EINVAL` – Invalid parameter.
- `-EALREADY` – Already in the given state.

enum *bt_mesh_feat_state* `bt_mesh_friend_get`(void)

Get the current Friend state.

Returns The Friend feature state.

Bluetooth Mesh Shell

The Bluetooth mesh shell subsystem provides a set of Bluetooth mesh shell commands for the *Shell* module. It allows for testing and exploring the Bluetooth mesh API through an interactive interface, without having to write an application.

The Bluetooth mesh shell interface provides access to most Bluetooth mesh features, including provisioning, configuration, and message sending.

Prerequisites The Bluetooth mesh shell subsystem depends on the *Configuration Client* and *Health Client* models.

Application The Bluetooth mesh shell subsystem is most easily used through the Bluetooth mesh shell application under `tests/bluetooth/mesh_shell`. See *Shell* for information on how to connect and interact with the Bluetooth mesh shell application.

Basic usage The Bluetooth mesh shell subsystem adds a single `mesh` command, which holds a set of sub-commands. Every time the device boots up, make sure to call `mesh init` before any of the other Bluetooth mesh shell commands can be called:

```
uart:~$ mesh init
```

Provisioning The mesh node must be provisioned to become part of the network. This is only necessary the first time the device boots up, as the device will remember its provisioning data between reboots.

The simplest way to provision the device is through self-provisioning. To provision the device with the default network key and address `0x0001`, execute:

```
uart:~$ mesh provision 0 0x0001
```

Since all mesh nodes use the same values for the default network key, this can be done on multiple devices, as long as they're assigned non-overlapping unicast addresses. Alternatively, to provision the device into an existing network, the unprovisioned beacon can be enabled with `mesh pb-adv on` or `mesh pb-gatt on`. The beacons can be picked up by an external provisioner, which can provision the node into its network.

Once the mesh node is part of a network, its transmission parameters can be controlled by the general configuration commands:

- To set the destination address, call `mesh dst <addr>`.
- To set the network key index, call `mesh netidx <NetIdx>`.
- To set the application key index, call `mesh appidx <AppIdx>`.

By default, the transmission parameters are set to send messages to the provisioned address and network key.

Configuration By setting the destination address to the local unicast address (`0x0001` in the `mesh provision` command above), we can perform self-configuration through any of the *Configuration Client model* commands.

A good first step is to read out the node's own composition data:

```
uart:~$ mesh get-comp
```

This prints a list of the composition data of the node, including a list of its model IDs.

Next, since the device has no application keys by default, it's a good idea to add one:

```
uart:~$ mesh app-key-add 0 0
```

Message sending With an application key added (see above), the mesh node's transition parameters are all valid, and the Bluetooth mesh shell can send raw mesh messages through the network.

For example, to send a Generic OnOff Set message, call:

```
uart:~$ mesh net-send 82020100
```

Note: All multibyte fields model messages are in little endian, except the opcode.

The message will be sent to the current destination address, using the current network and application key indexes. As the destination address points to the local unicast address by default, the device will only send packets to itself. To change the destination address to the All Nodes broadcast address, call:

```
uart:~$ mesh dst 0xffff
```

With the destination address set to `0xffff`, any other mesh nodes in the network with the configured network and application keys will receive and process the messages we send.

Note: To change the configuration of the device, the destination address must be set back to the local unicast address before issuing any configuration commands.

Sending raw mesh packets is a good way to test model message handler implementations during development, as it can be done without having to implement the sending model. By default, only the reception of the model messages can be tested this way, as the Bluetooth mesh shell only includes the foundation

models. To receive a packet in the mesh node, you have to add a model with a valid opcode handler list to the composition data in `subsys/bluetooth/mesh/shell.c`, and print the incoming message to the shell in the handler callback.

Parameter formats The Bluetooth mesh shell commands are parsed with a variety of formats:

Table 3: Parameter formats

Type	Description	Example
Integers	The default format unless something else is specified. Can be either decimal or hexadecimal.	`1234`, `0xabcd01234`
Hexstrings	For raw byte arrays, like UUIDs, key values and message payloads, the parameters should be formatted as an unbroken string of hexadecimal values without any prefix.	`deadbeef01234`
Booleans	Boolean values are denoted in the API documentation as `<val: on, off>`.	`on`, `off`, `enabled`, `disabled`, `1`, `0`

Commands The Bluetooth mesh shell implements a large set of commands. Some of the commands accept parameters, which are mentioned in brackets after the command name. For example, `mesh lpn <value: off, on>`. Mandatory parameters are marked with angle brackets (e.g. `<NetKeyIndex>`), and optional parameters are marked with square brackets (e.g. `[destination address]`).

The Bluetooth mesh shell commands are divided into the following groups:

Note: Some commands depend on specific features being enabled in the compile time configuration of the application. Not all features are enabled by default. The list of available Bluetooth mesh shell commands can be shown in the shell by calling `mesh` without any arguments.

General configuration

`mesh init`

Initialize the mesh. This command must be run before any other mesh command.

`mesh reset <addr>`

reset the local mesh node to its initial unprovisioned state or reset a remote node and remove it from the network. * `addr`: address of the node to reset.

`mesh lpn <value: off, on>`

Enable or disable Low Power operation. Once enabled, the device will turn off its radio and start polling for friend nodes. The device will not be able to receive messages from the mesh network until the friendship has been established.

- `value`: Sets whether Low Power operation is enabled.

`mesh poll`

Perform a poll to the friend node, to receive any pending messages. Only available when LPN is enabled.

`mesh ident`

Enable the Proxy Node Identity beacon, allowing Proxy devices to connect explicitly to this device. The beacon will run for 60 seconds before the node returns to normal Proxy beacons.

`mesh dst [destination address]`

Get or set the message destination address. The destination address determines where mesh packets are sent with the shell, but has no effect on modules outside the shell's control.

- `destination address`: If present, sets the new 16-bit mesh destination address. If omitted, the current destination address is printed.

`mesh netidx [NetIdx]`

Get or set the message network index. The network index determines which network key is used to encrypt mesh packets that are sent with the shell, but has no effect on modules outside the shell's control. The network key must already be added to the device, either through provisioning or by a Configuration Client.

- `NetIdx`: If present, sets the new network index. If omitted, the current network index is printed.

`mesh appidx [AppIdx]`

Get or set the message application index. The application index determines which application key is used to encrypt mesh packets that are sent with the shell, but has no effect on modules outside the shell's control. The application key must already be added to the device by a Configuration Client, and must be bound to the current network index.

- `AppIdx`: If present, sets the new application index. If omitted, the current application index is printed.

`mesh net-send <hex string>`

Send a raw mesh message with the current destination address, network and application index. The message opcode must be encoded manually.

- `hex string` Raw hexadecimal representation of the message to send.

Testing

`mesh iv-update`

Force an IV update.

`mesh iv-update-test <value: off, on>`

Set the IV update test mode. In test mode, the IV update timing requirements are bypassed.

- `value`: Enable or disable the IV update test mode.

`mesh rpl-clear`

Clear the replay protection list, forcing the node to forget all received messages.

Warning: Clearing the replay protection list breaks the security mechanisms of the mesh node, making it susceptible to message replay attacks. This should never be performed in a real deployment.

Provisioning

`mesh pb-gatt <val: off, on>`

Start or stop advertising a connectable unprovisioned beacon. The connectable unprovisioned beacon allows the mesh node to be discovered by nearby GATT based provisioners, and provisioned through the GATT bearer.

- `val`: Enable or disable provisioning with GATT

`mesh pb-adv <val: off, on>`

Start or stop advertising the unprovisioned beacon. The unprovisioned beacon allows the mesh node to be discovered by nearby advertising-based provisioners, and provisioned through the advertising bearer.

- `val`: Enable or disable provisioning with advertiser

`mesh provision-adv <UUID> <NetKeyIndex> <addr> <AttentionDuration>`

Provision a nearby device into the mesh. The mesh node starts scanning for unprovisioned beacons with the given UUID. Once found, the unprovisioned device will be added to the mesh network with the given unicast address, and given the network key indicated by `NetKeyIndex`.

- `UUID`: UUID of the unprovisioned device.
- `NetKeyIndex`: Index of the network key to pass to the device.
- `addr`: First unicast address to assign to the unprovisioned device. The device will occupy as many addresses as it has elements, and all must be available.
- `AttentionDuration`: The duration in seconds the unprovisioned device will identify itself for, if supported. See *Attention state* for details.

`mesh uuid <UUID: 1-16 hex values>`

Set the mesh node's UUID, used in the unprovisioned beacons.

- `UUID`: New 128-bit UUID value. Any missing bytes will be zero.

`mesh input-num <number>`

Input a numeric OOB authentication value. Only valid when prompted by the shell during provisioning. The input number must match the number presented by the other participant in the provisioning.

- `number`: Decimal authentication number.

`mesh input-str <string>`

Input an alphanumeric OOB authentication value. Only valid when prompted by the shell during provisioning. The input string must match the string presented by the other participant in the provisioning.

- `string`: Unquoted alphanumeric authentication string.

`mesh static-oob [val: 1-16 hex values]`

Set or clear the static OOB authentication value. The static OOB authentication value must be set before provisioning starts to have any effect. The static OOB value must be same on both participants in the provisioning.

- `val`: If present, indicates the new hexadecimal value of the static OOB. If omitted, the static OOB value is cleared.

`mesh provision <NetKeyIndex> <addr> [IVIndex]`

Provision the mesh node itself. If the Configuration database is enabled, the network key must be created. Otherwise, the default key value is used.

- `NetKeyIndex`: Index of the network key to provision.
- `addr`: First unicast address to assign to the device. The device will occupy as many addresses as it has elements, and all must be available.
- `IVindex`: Indicates the current network IV index. Defaults to 0 if omitted.

`mesh beacon-listen <val: off, on>`

Enable or disable printing of incoming unprovisioned beacons. Allows a provisioner device to detect nearby unprovisioned devices and provision them.

- `val`: Whether to enable the unprovisioned beacon printing.

Configuration Client model The Bluetooth mesh shell module instantiates a Configuration Client model for configuring itself and other nodes in the mesh network.

The Configuration Client uses the general messages parameters set by `mesh dst` and `mesh netidx` to target specific nodes. When the Bluetooth mesh shell node is provisioned, the Configuration Client model targets itself by default. When another node has been provisioned by the Bluetooth mesh shell, the Configuration Client model targets the new node. The Configuration Client always sends messages using the Device key bound to the destination address, so it will only be able to configure itself and mesh nodes it provisioned.

`mesh timeout [timeout in seconds]`

Get and set the Config Client model timeout used during message sending.

- `timeout in seconds`: If present, set the Config Client model timeout in seconds. If omitted, the current timeout is printed.

`mesh get-comp [page]`

Read a composition data page. The full composition data page will be printed. If the target does not have the given page, it will return the last page before it.

- `page`: The composition data page to request. Defaults to 0 if omitted.

`mesh beacon [val: off, on]`

Get or set the network beacon transmission.

- `val`: If present, enables or disables sending of the network beacon. If omitted, the current network beacon state is printed.

`mesh ttl [ttl: 0x00, 0x02-0x7f]`

Get or set the default TTL value.

- `ttl`: If present, sets the new default TTL value. If omitted, the current default TTL value is printed.

`mesh friend [val: off, on]`

Get or set the Friend feature.

- `val`: If present, enables or disables the Friend feature. If omitted, the current Friend feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.

`mesh gatt-proxy [val: off, on]`

Get or set the GATT Proxy feature.

- `val`: If present, enables or disables the GATT Proxy feature. If omitted, the current GATT Proxy feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.

`mesh relay [<val: off, on> [<count: 0-7> [interval: 10-320]]]`

Get or set the Relay feature and its parameters.

- `val`: If present, enables or disables the Relay feature. If omitted, the current Relay feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.
- `count`: Sets the new relay retransmit count if `val` is `on`. Ignored if `val` is `off`. Defaults to 2 if omitted.
- `interval`: Sets the new relay retransmit interval in milliseconds if `val` is `on`. Ignored if `val` is `off`. Defaults to 20 if omitted.

`mesh net-transmit-param [<count: 0-7> <interval: 10-320>]`

Get or set the network transmit parameters.

- `count`: Sets the number of additional network transmits for every sent message.
- `interval`: Sets the new network retransmit interval in milliseconds.

`mesh net-key-add <NetKeyIndex> [val]`

Add a network key to the target node. Adds the key to the Configuration Database if enabled.

- `NetKeyIndex`: The network key index to add.
- `val`: If present, sets the key value as a 128-bit hexadecimal value. Any missing bytes will be zero. Only valid if the key does not already exist in the Configuration Database. If omitted, the default key value is used.

`mesh net-key-get`

Get a list of known network key indexes.

`mesh net-key-del <NetKeyIndex>`

Delete a network key from the target node.

- `NetKeyIndex`: The network key index to delete.

`mesh app-key-add <NetKeyIndex> <AppKeyIndex> [val]`

Add an application key to the target node. Adds the key to the Configuration Database if enabled.

- `NetKeyIndex`: The network key index the application key is bound to.
- `AppKeyIndex`: The application key index to add.
- `val`: If present, sets the key value as a 128-bit hexadecimal value. Any missing bytes will be zero. Only valid if the key does not already exist in the Configuration Database. If omitted, the default key value is used.

`mesh app-key-get <NetKeyIndex>`

Get a list of known application key indexes bound to the given network key index.

- `NetKeyIndex`: Network key indexes to get a list of application key indexes from.

`mesh app-key-del <NetKeyIndex> <AppKeyIndex>`

Delete an application key from the target node.

- `NetKeyIndex`: The network key index the application key is bound to.
- `AppKeyIndex`: The application key index to delete.

`mesh mod-app-bind <addr> <AppIndex> <Model ID> [Company ID]`

Bind an application key to a model. Models can only encrypt and decrypt messages sent with application keys they are bound to.

- `addr`: Address of the element the model is on.
- `AppIndex`: The application key to bind to the model.
- `Model ID`: The model ID of the model to bind the key to.
- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh mod-app-unbind <addr> <AppIndex> <Model ID> [Company ID]`

Unbind an application key from a model.

- `addr`: Address of the element the model is on.
- `AppIndex`: The application key to unbind from the model.
- `Model ID`: The model ID of the model to unbind the key from.
- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh mod-app-get <elem addr> <Model ID> [Company ID]`

Get a list of application keys bound to a model.

- `elem addr`: Address of the element the model is on.
- `Model ID`: The model ID of the model to get the bound keys of.
- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh mod-pub <addr> <mod id> [cid] [<PubAddr> <AppKeyIndex> <cred: off, on> <ttl> <period> <count> <interval>]`

Get or set the publication parameters of a model. If all publication parameters are included, they become the new publication parameters of the model. If all publication parameters are omitted, print the current publication parameters of the model.

- `addr`: Address of the element the model is on.
- `Model ID`: The model ID of the model to get the bound keys of.
- `cid`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

Publication parameters:

- `PubAddr`: The destination address to publish to.
- `AppKeyIndex`: The application key index to publish with.
- `cred`: Whether to publish with Friendship credentials when acting as a Low Power Node.
- `ttl`: TTL value to publish with (`0x00` to `0x07f`).
- `period`: Encoded publication period, or 0 to disable periodic publication.
- `count`: Number of retransmission for each published message (`0` to `7`).
- `interval` The interval between each retransmission, in milliseconds. Must be a multiple of 50.

`mesh mod-sub-add <elem addr> <sub addr> <Model ID> [Company ID]`

Subscription the model to a group address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `elem addr`: Address of the element the model is on.
- `sub addr`: 16-bit group address the model should subscribe to (`0xc000` to `0xFEFF`).
- `Model ID`: The model ID of the model to add the subscription to.
- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh mod-sub-del <elem addr> <sub addr> <Model ID> [Company ID]`

Unsubscribe a model from a group address.

- `elem addr`: Address of the element the model is on.
- `sub addr`: 16-bit group address the model should remove from its subscription list (`0xc000` to `0xFEFF`).
- `Model ID`: The model ID of the model to add the subscription to.
- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh mod-sub-add-va <elem addr> <Label UUID> <Model ID> [Company ID]`

Subscribe the model to a virtual address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `elem addr`: Address of the element the model is on.
- `Label UUID`: 128-bit label UUID of the virtual address to subscribe to. Any omitted bytes will be zero.
- `Model ID`: The model ID of the model to add the subscription to.
- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh mod-sub-del-va <elem addr> <Label UUID> <Model ID> [Company ID]`

Unsubscribe a model from a virtual address.

- `elem addr`: Address of the element the model is on.
- `Label UUID`: 128-bit label UUID of the virtual address to remove the subscribtion of. Any omitted bytes will be zero.
- `Model ID`: The model ID of the model to add the subscription to.
- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh mod-sub-get <elem addr> <Model ID> [Company ID]`

Get a list of addresses the model subscribes to.

- `elem addr`: Address of the element the model is on.
- `Model ID`: The model ID of the model to get the subscription list of.

- `Company ID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh hb-sub [<src> <dst> <period>]`

Get or set the Heartbeat subscription parameters. A node only receives Heartbeat messages matching the Heartbeat subscription parameters. Sets the Heartbeat subscription parameters if present, or prints the current Heartbeat subscription parameters if called with no parameters.

- `src`: Unicast source address to receive Heartbeat messages from.
- `dst`: Destination address to receive Heartbeat messages on.
- `period`: Logarithmic representation of the Heartbeat subscription period:
  - `0`: Heartbeat subscription will be disabled.
  - `1` to `17`: The node will subscribe to Heartbeat messages for $2^{(\text{period} - 1)}$ seconds.

`mesh hb-pub [<dst> <count> <period> <ttl> <features> <NetKeyIndex>]`

Get or set the Heartbeat publication parameters. Sets the Heartbeat publication parameters if present, or prints the current Heartbeat publication parameters if called with no parameters.

- `dst`: Destination address to publish Heartbeat messages to.
- `count`: Logarithmic representation of the number of Heartbeat messages to publish periodically:
  - `0`: Heartbeat messages are not published periodically.
  - `1` to `17`: The node will periodically publish $2^{(\text{count} - 1)}$ Heartbeat messages.
  - `255`: Heartbeat messages will be published periodically indefinitely.
- `period`: Logarithmic representation of the Heartbeat publication period:
  - `0`: Heartbeat messages are not published periodically.
  - `1` to `17`: The node will publish Heartbeat messages every $2^{(\text{period} - 1)}$ seconds.
- `ttl`: The TTL value to publish Heartbeat messages with (`0x00` to `0x7f`).
- `features`: Bitfield of features that should trigger a Heartbeat publication when changed:
  - `Bit 0`: Relay feature.
  - `Bit 1`: Proxy feature.
  - `Bit 2`: Friend feature.
  - `Bit 3`: Low Power feature.
- `NetKeyIndex`: Index of the network key to publish Heartbeat messages with.

Health Client model The Bluetooth mesh shell module instantiates a Health Client model for configuring itself and other nodes in the mesh network.

The Health Client uses the general messages parameters set by `mesh dst` and `mesh netidx` to target specific nodes. When the Bluetooth mesh shell node is provisioned, the Health Client model targets itself by default. When another node has been provisioned by the Bluetooth mesh shell, the Health Client model targets the new node. The Health Client always sends messages using the Device key bound to the destination address, so it will only be able to configure itself and mesh nodes it provisioned.

`mesh fault-get <Company ID>`

Get a list of registered faults for a Company ID.

- `Company ID`: Company ID to get faults for.

`mesh fault-clear <Company ID>`

Clear the list of faults for a Company ID.

- `Company ID`: Company ID to clear the faults for.

`mesh fault-clear-unack <Company ID>`

Clear the list of faults for a Company ID without requesting a response.

- `Company ID`: Company ID to clear the faults for.

`mesh fault-test <Company ID> <Test ID>`

Invoke a self-test procedure, and show a list of triggered faults.

- `Company ID`: Company ID to perform self-tests for.
- `Test ID`: Test to perform.

`mesh fault-test-unack <Company ID> <Test ID>`

Invoke a self-test procedure without requesting a response.

- `Company ID`: Company ID to perform self-tests for.
- `Test ID`: Test to perform.

`mesh period-get`

Get the current Health Server publish period divisor.

`mesh period-set <divisor>`

Set the current Health Server publish period divisor. When a fault is detected, the Health Server will start publishing is fault status with a reduced interval. The reduced interval is determined by the Health Server publish period divisor: Fault publish period = Publish period / 2^{divisor}.

- `divisor`: The new Health Server publish period divisor.

`mesh period-set-unack <divisor>`

Set the current Health Server publish period divisor. When a fault is detected, the Health Server will start publishing is fault status with a reduced interval. The reduced interval is determined by the Health Server publish period divisor: Fault publish period = Publish period / 2^{divisor}.

- `divisor`: The new Health Server publish period divisor.

`mesh attention-get`

Get the current Health Server attention state.

`mesh attention-set <timer>`

Enable the Health Server attention state for some time.

- `timer`: Duration of the attention state, in seconds (`0` to `255`)

`mesh attention-set-unack <timer>`

Enable the Health Server attention state for some time without requesting a response.

- `timer`: Duration of the attention state, in seconds (`0` to `255`)

Health Server model

`mesh add-fault <Fault ID>`

Register a new Fault for the Linux Foundation Company ID.

- `Fault ID`: ID of the fault to register (`0x0001` to `0xFFFF`)

`mesh del-fault [Fault ID]`

Remove registered faults for the Linux Foundation Company ID.

- `Fault ID`: If present, the given fault ID will be deleted. If omitted, all registered faults will be cleared.

Configuration database The Configuration database is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_CDB` configuration option. The Configuration database is only available on provisioner devices, and allows them to store all information about the mesh network. To avoid conflicts, there should only be one mesh node in the network with the Configuration database enabled. This node is the Configurator, and is responsible for adding new nodes to the network and configuring them.

`mesh cdb-create [NetKey]`

Create a Configuration database.

- `NetKey`: Optional network key value of the primary network key (NetKeyIndex=0). Defaults to the default key value if omitted.

`mesh cdb-clear`

Clear all data from the Configuration database.

`mesh cdb-show`

Show all data in the Configuration database.

`mesh cdb-node-add <UUID> <addr> <num-elem> <NetKeyIdx> [DevKey]`

Manually add a mesh node to the configuration database. Note that devices provisioned with `mesh provision` and `mesh provision-adv` will be added automatically if the Configuration Database is enabled and created.

- `UUID`: 128-bit hexadecimal UUID of the node. Any omitted bytes will be zero.
- `addr`: Unicast address of the node, or 0 to automatically choose the lowest available address.

- `num-elem`: Number of elements on the node.
- `NetKeyIdx`: The network key the node was provisioned with.
- `DevKey`: Optional 128-bit device key value for the device. If omitted, a random value will be generated.

`mesh cdb-node-del <addr>`

Delete a mesh node from the Configuration database. If possible, the node should be reset with `mesh reset` before it is deleted from the Configuration database, to avoid unexpected behavior and uncontrolled access to the network.

- `addr` Address of the node to delete.

`mesh cdb-subnet-add <NeyKeyIdx> [<NetKey>]`

Add a network key to the Configuration database. The network key can later be passed to mesh nodes in the network. Note that adding a key to the Configuration database does not automatically add it to the local node's list of known network keys.

- `NetKeyIdx`: Key index of the network key to add.
- `NetKey`: Optional 128-bit network key value. Any missing bytes will be zero. If omitted, a random value will be generated.

`mesh cdb-subnet-del <NetKeyIdx>`

Delete a network key from the Configuration database.

- `NetKeyIdx`: Key index of the network key to delete.

`mesh cdb-app-key-add <NetKeyIdx> <AppKeyIdx> [<AppKey>]`

Add an application key to the Configuration database. The application key can later be passed to mesh nodes in the network. Note that adding a key to the Configuration database does not automatically add it to the local node's list of known application keys.

- `NetKeyIdx`: Network key index the application key is bound to.
- `AppKeyIdx`: Key index of the application key to add.
- `AppKey`: Optional 128-bit application key value. Any missing bytes will be zero. If omitted, a random value will be generated.

`mesh cdb-app-key-del <AppKeyIdx>`

Delete an application key from the Configuration database.

- `AppKeyIdx`: Key index of the application key to delete.

7.4.12 Serial Port Emulation (RFCOMM)

API Reference

group `bt_rfcomm`

RFCOMM.

Typedefs

typedef enum *bt_rfcomm_role* `bt_rfcomm_role_t`

Role of RFCOMM session and dlc. Used only by internal APIs.

Enums

enum `[anonymous]`

Values:

enumerator `BT_RFCOMM_CHAN_HFP_HF` = 1

enumerator `BT_RFCOMM_CHAN_HFP_AG`

enumerator `BT_RFCOMM_CHAN_HSP_AG`

enumerator `BT_RFCOMM_CHAN_HSP_HS`

enumerator `BT_RFCOMM_CHAN_SPP`

enum `bt_rfcomm_role`

Role of RFCOMM session and dlc. Used only by internal APIs.

Values:

enumerator `BT_RFCOMM_ROLE_ACCEPTOR`

enumerator `BT_RFCOMM_ROLE_INITIATOR`

Functions

int `bt_rfcomm_server_register`(struct *bt_rfcomm_server* *server)

Register RFCOMM server.

Register RFCOMM server for a channel, each new connection is authorized using the accept() callback which in case of success shall allocate the dlc structure to be used by the new connection.

Parameters

- `server` – Server structure.

Returns 0 in case of success or negative value in case of error.

int `bt_rfcomm_dlc_connect`(struct bt_conn *conn, struct *bt_rfcomm_dlc* *dlc, uint8_t channel)

Connect RFCOMM channel.

Connect RFCOMM dlc by channel, once the connection is completed dlc connected() callback will be called. If the connection is rejected disconnected() callback is called instead.

Parameters

- `conn` – Connection object.
- `dlc` – Dlc object.

- `channel` – Server channel to connect to.

Returns 0 in case of success or negative value in case of error.

int `bt_rfcomm_dlc_send`(struct *bt_rfcomm_dlc* *dlc, struct *net_buf* *buf)

Send data to RFCOMM.

Send data from buffer to the dlc. Length should be less than or equal to mtu.

Parameters

- `dlc` – Dlc object.
- `buf` – Data buffer.

Returns Bytes sent in case of success or negative value in case of error.

int `bt_rfcomm_dlc_disconnect`(struct *bt_rfcomm_dlc* *dlc)

Disconnect RFCOMM dlc.

Disconnect RFCOMM dlc, if the connection is pending it will be canceled and as a result the dlc disconnected() callback is called.

Parameters

- `dlc` – Dlc object.

Returns 0 in case of success or negative value in case of error.

struct *net_buf* *`bt_rfcomm_create_pdu`(struct *net_buf_pool* *pool)

Allocate the buffer from pool after reserving head room for RFCOMM, L2CAP and ACL headers.

Parameters

- `pool` – Which pool to take the buffer from.

Returns New buffer.

struct `bt_rfcomm_dlc_ops`

#include <rfcomm.h> RFCOMM DLC operations structure.

Public Members

void (*`connected`)(struct *bt_rfcomm_dlc* *dlc)

DLC connected callback

If this callback is provided it will be called whenever the connection completes.
Param dlc The dlc that has been connected

void (*`disconnected`)(struct *bt_rfcomm_dlc* *dlc)

DLC disconnected callback

If this callback is provided it will be called whenever the dlc is disconnected, including when a connection gets rejected or cancelled (both incoming and outgoing)
Param dlc The dlc that has been Disconnected

void (*`recv`)(struct *bt_rfcomm_dlc* *dlc, struct *net_buf* *buf)

DLC recv callback
Param dlc The dlc receiving data.
Param buf Buffer containing incoming data.

struct bt_rfcomm_dlc

#include <rfcomm.h> RFCOMM DLC structure.

struct bt_rfcomm_server

#include <rfcomm.h>

Public Members

uint8_t channel

Server Channel

int (*accept)(struct bt_conn *conn, struct *bt_rfcomm_dlc* **dlc)

Server accept callback

This callback is called whenever a new incoming connection requires authorization.

Param conn The connection that is requesting authorization
Param dlc Pointer to received the allocated dlc
Return 0 in case of success or negative value in case of error.

7.4.13 Service Discovery Protocol (SDP)

API Reference

group bt_sdp

Service Discovery Protocol (SDP)

Defines

BT_SDP_SDP_SERVER_SVCLASS

BT_SDP_BROWSE_GRP_DESC_SVCLASS

BT_SDP_PUBLIC_BROWSE_GROUP

BT_SDP_SERIAL_PORT_SVCLASS

BT_SDP_LAN_ACCESS_SVCLASS

BT_SDP_DIALUP_NET_SVCLASS

BT_SDP_IRMC_SYNC_SVCLASS

BT_SDP_OBEX_OBJPUSH_SVCLASS

BT_SDP_OBEX_FILETRANS_SVCLASS

BT_SDP_IRMC_SYNC_CMD_SVCLASS

BT_SDP_HEADSET_SVCLASS

BT_SDP_CORDLESS_TELEPHONY_SVCLASS

BT_SDP_AUDIO_SOURCE_SVCLASS

BT_SDP_AUDIO_SINK_SVCLASS

BT_SDP_AV_REMOTE_TARGET_SVCLASS

BT_SDP_ADVANCED_AUDIO_SVCLASS

BT_SDP_AV_REMOTE_SVCLASS

BT_SDP_AV_REMOTE_CONTROLLER_SVCLASS

BT_SDP_INTERCOM_SVCLASS

BT_SDP_FAX_SVCLASS

BT_SDP_HEADSET_AGW_SVCLASS

BT_SDP_WAP_SVCLASS

BT_SDP_WAP_CLIENT_SVCLASS

BT_SDP_PANU_SVCLASS

BT_SDP_NAP_SVCLASS

BT_SDP_GN_SVCLASS

BT_SDP_DIRECT_PRINTING_SVCLASS

BT_SDP_REFERENCE_PRINTING_SVCLASS

BT_SDP_IMAGING_SVCLASS

BT_SDP_IMAGING_RESPONDER_SVCLASS

BT_SDP_IMAGING_ARCHIVE_SVCLASS

BT_SDP_IMAGING_REFOBJS_SVCLASS

BT_SDP_HANDSFREE_SVCLASS

BT_SDP_HANDSFREE_AGW_SVCLASS

BT_SDP_DIRECT_PRT_REFOBJS_SVCLASS

BT_SDP_REFLECTED_UI_SVCLASS

BT_SDP_BASIC_PRINTING_SVCLASS

BT_SDP_PRINTING_STATUS_SVCLASS

BT_SDP_HID_SVCLASS

BT_SDP_HCR_SVCLASS

BT_SDP_HCR_PRINT_SVCLASS

BT_SDP_HCR_SCAN_SVCLASS

BT_SDP_CIP_SVCLASS

BT_SDP_VIDEO_CONF_GW_SVCLASS

BT_SDP_UDI_MT_SVCLASS

BT_SDP_UDI_TA_SVCLASS

BT_SDP_AV_SVCLASS

BT_SDP_SAP_SVCLASS

BT_SDP_PBAP_PCE_SVCLASS

BT_SDP_PBAP_PSE_SVCLASS

BT_SDP_PBAP_SVCLASS

BT_SDP_MAP_MSE_SVCLASS

BT_SDP_MAP_MCE_SVCLASS

BT_SDP_MAP_SVCLASS

BT_SDP_GNSS_SVCLASS

BT_SDP_GNSS_SERVER_SVCLASS

BT_SDP_MPS_SC_SVCLASS

BT_SDP_MPS_SVCLASS

BT_SDP_PNP_INFO_SVCLASS

BT_SDP_GENERIC_NETWORKING_SVCLASS

BT_SDP_GENERIC_FILETRANS_SVCLASS

BT_SDP_GENERIC_AUDIO_SVCLASS

BT_SDP_GENERIC_TELEPHONY_SVCLASS

BT_SDP_UPNP_SVCLASS

BT_SDP_UPNP_IP_SVCLASS

BT_SDP_UPNP_PAN_SVCLASS

BT_SDP_UPNP_LAP_SVCLASS

BT_SDP_UPNP_L2CAP_SVCLASS

BT_SDP_VIDEO_SOURCE_SVCLASS

BT_SDP_VIDEO_SINK_SVCLASS

BT_SDP_VIDEO_DISTRIBUTION_SVCLASS

BT_SDP_HDP_SVCLASS

BT_SDP_HDP_SOURCE_SVCLASS

BT_SDP_HDP_SINK_SVCLASS

BT_SDP_GENERIC_ACCESS_SVCLASS

BT_SDP_GENERIC_ATTRIB_SVCLASS

BT_SDP_APPLE_AGENT_SVCLASS

BT_SDP_SERVER_RECORD_HANDLE

BT_SDP_ATTR_RECORD_HANDLE

BT_SDP_ATTR_SVCLASS_ID_LIST

BT_SDP_ATTR_RECORD_STATE

BT_SDP_ATTR_SERVICE_ID

BT_SDP_ATTR_PROTO_DESC_LIST

BT_SDP_ATTR_BROWSE_GRP_LIST

BT_SDP_ATTR_LANG_BASE_ATTR_ID_LIST

BT_SDP_ATTR_SVCINFO_TTL

BT_SDP_ATTR_SERVICE_AVAILABILITY

BT_SDP_ATTR_PROFILE_DESC_LIST

BT_SDP_ATTR_DOC_URL

BT_SDP_ATTR_CLNT_EXEC_URL

BT_SDP_ATTR_ICON_URL

BT_SDP_ATTR_ADD_PROTO_DESC_LIST

BT_SDP_ATTR_GROUP_ID

BT_SDP_ATTR_IP_SUBNET

BT_SDP_ATTR_VERSION_NUM_LIST

BT_SDP_ATTR_SUPPORTED_FEATURES_LIST

BT_SDP_ATTR_GOEP_L2CAP_PSM

BT_SDP_ATTR_SVCDB_STATE

BT_SDP_ATTR_MPSD_SCENARIOS

BT_SDP_ATTR_MPMD_SCENARIOS

BT_SDP_ATTR_MPS_DEPENDENCIES

BT_SDP_ATTR_SERVICE_VERSION

BT_SDP_ATTR_EXTERNAL_NETWORK

BT_SDP_ATTR_SUPPORTED_DATA_STORES_LIST

BT_SDP_ATTR_DATA_EXCHANGE_SPEC

BT_SDP_ATTR_NETWORK

BT_SDP_ATTR_FAX_CLASS1_SUPPORT

BT_SDP_ATTR_REMOTE_AUDIO_VOLUME_CONTROL

BT_SDP_ATTR_MCAP_SUPPORTED_PROCEDURES

BT_SDP_ATTR_FAX_CLASS20_SUPPORT

BT_SDP_ATTR_SUPPORTED_FORMATS_LIST

BT_SDP_ATTR_FAX_CLASS2_SUPPORT

BT_SDP_ATTR_AUDIO_FEEDBACK_SUPPORT

BT_SDP_ATTR_NETWORK_ADDRESS

BT_SDP_ATTR_WAP_GATEWAY

BT_SDP_ATTR_HOMEPAGE_URL

BT_SDP_ATTR_WAP_STACK_TYPE

BT_SDP_ATTR_SECURITY_DESC

BT_SDP_ATTR_NET_ACCESS_TYPE

BT_SDP_ATTR_MAX_NET_ACCESSRATE

BT_SDP_ATTR_IP4_SUBNET

BT_SDP_ATTR_IP6_SUBNET

BT_SDP_ATTR_SUPPORTED_CAPABILITIES

BT_SDP_ATTR_SUPPORTED_FEATURES

BT_SDP_ATTR_SUPPORTED_FUNCTIONS

BT_SDP_ATTR_TOTAL_IMAGING_DATA_CAPACITY

BT_SDP_ATTR_SUPPORTED_REPOSITORIES

BT_SDP_ATTR_MAS_INSTANCE_ID

BT_SDP_ATTR_SUPPORTED_MESSAGE_TYPES

BT_SDP_ATTR_PBAP_SUPPORTED_FEATURES

BT_SDP_ATTR_MAP_SUPPORTED_FEATURES

BT_SDP_ATTR_SPECIFICATION_ID

BT_SDP_ATTR_VENDOR_ID

BT_SDP_ATTR_PRODUCT_ID

BT_SDP_ATTR_VERSION

BT_SDP_ATTR_PRIMARY_RECORD

BT_SDP_ATTR_VENDOR_ID_SOURCE

BT_SDP_ATTR_HID_DEVICE_RELEASE_NUMBER

BT_SDP_ATTR_HID_PARSER_VERSION

BT_SDP_ATTR_HID_DEVICE_SUBCLASS

BT_SDP_ATTR_HID_COUNTRY_CODE

BT_SDP_ATTR_HID_VIRTUAL_CABLE

BT_SDP_ATTR_HID_RECONNECT_INITIATE

BT_SDP_ATTR_HID_DESCRIPTOR_LIST

BT_SDP_ATTR_HID_LANG_ID_BASE_LIST

BT_SDP_ATTR_HID_SDP_DISABLE

BT_SDP_ATTR_HID_BATTERY_POWER

BT_SDP_ATTR_HID_REMOTE_WAKEUP

BT_SDP_ATTR_HID_PROFILE_VERSION

BT_SDP_ATTR_HID_SUPERVISION_TIMEOUT

BT_SDP_ATTR_HID_NORMALLY_CONNECTABLE

BT_SDP_ATTR_HID_BOOT_DEVICE

BT_SDP_PRIMARY_LANG_BASE

BT_SDP_ATTR_SVCNAME_PRIMARY

BT_SDP_ATTR_SVCDESC_PRIMARY

BT_SDP_ATTR_PROVNAME_PRIMARY

BT_SDP_DATA_NIL

BT_SDP_UINT8

BT_SDP_UINT16

BT_SDP_UINT32

BT_SDP_UINT64

BT_SDP_UINT128

BT_SDP_INT8

BT_SDP_INT16

BT_SDP_INT32

BT_SDP_INT64

BT_SDP_INT128

BT_SDP_UUID_UNSPEC

BT_SDP_UUID16

BT_SDP_UUID32

BT_SDP_UUID128

BT_SDP_TEXT_STR_UNSPEC

BT_SDP_TEXT_STR8

BT_SDP_TEXT_STR16

BT_SDP_TEXT_STR32

BT_SDP_BOOL

BT_SDP_SEQ_UNSPEC

BT_SDP_SEQ8

BT_SDP_SEQ16

BT_SDP_SEQ32

BT_SDP_ALT_UNSPEC

BT_SDP_ALT8

BT_SDP_ALT16

BT_SDP_ALT32

BT_SDP_URL_STR_UNSPEC

BT_SDP_URL_STR8

BT_SDP_URL_STR16

BT_SDP_URL_STR32

BT_SDP_TYPE_DESC_MASK

BT_SDP_SIZE_DESC_MASK

BT_SDP_SIZE_INDEX_OFFSET

BT_SDP_ARRAY_8(...)
: Declare an array of 8-bit elements in an attribute.

BT_SDP_ARRAY_16(...)
: Declare an array of 16-bit elements in an attribute.

BT_SDP_ARRAY_32(...)

Declare an array of 32-bit elements in an attribute.

BT_SDP_TYPE_SIZE(_type)

Declare a fixed-size data element header.

Parameters

- _type – Data element header containing type and size descriptors.

BT_SDP_TYPE_SIZE_VAR(_type, _size)

Declare a variable-size data element header.

Parameters

- _type – Data element header containing type and size descriptors.
- _size – The actual size of the data.

BT_SDP_DATA_ELEM_LIST(...)

Declare a list of data elements.

BT_SDP_NEW_SERVICE

SDP New Service Record Declaration Macro.

Helper macro to declare a new service record. Default attributes: Record Handle, Record State, Language Base, Root Browse Group

BT_SDP_LIST(_att_id, _type_size, _data_elem_seq)

Generic SDP List Attribute Declaration Macro.

Helper macro to declare a list attribute.

Parameters

- _att_id – List Attribute ID.
- _data_elem_seq – Data element sequence for the list.
- _type_size – SDP type and size descriptor.

BT_SDP_SERVICE_ID(_uuid)

SDP Service ID Attribute Declaration Macro.

Helper macro to declare a service ID attribute.

Parameters

- _uuid – Service ID 16bit UUID.

BT_SDP_SERVICE_NAME(_name)

SDP Name Attribute Declaration Macro.

Helper macro to declare a service name attribute.

Parameters

- _name – Service name as a string (up to 256 chars).

BT_SDP_SUPPORTED_FEATURES(_features)

SDP Supported Features Attribute Declaration Macro.

Helper macro to declare supported features of a profile/protocol.

Parameters

- _features – Feature mask as 16bit unsigned integer.

BT_SDP_RECORD(_attrs)

SDP Service Declaration Macro.

Helper macro to declare a service.

Parameters

- `_attrs` – List of attributes for the service record.

Typedefs

typedef uint8_t (*bt_sdp_discover_func_t)(struct bt_conn *conn, struct *bt_sdp_client_result* *result)

Callback type reporting to user that there is a resolved result on remote for given UUID and the result record buffer can be used by user for further inspection.

A function of this type is given by the user to the *bt_sdp_discover_params* object. It'll be called on each valid record discovery completion for given UUID. When UUID resolution gives back no records then NULL is passed to the user. Otherwise user can get valid record(s) and then the internal hint 'next record' is set to false saying the UUID resolution is complete or the hint can be set by caller to true meaning that next record is available for given UUID. The returned function value allows the user to control retrieving follow-up resolved records if any. If the user doesn't want to read more resolved records for given UUID since current record data fulfills its requirements then should return BT_SDP_DISCOVER_UUID_STOP. Otherwise returned value means more subcall iterations are allowable.

Param conn Connection object identifying connection to queried remote.

Param result Object pointing to logical unparsed SDP record collected on base of response driven by given UUID.

Return BT_SDP_DISCOVER_UUID_STOP in case of no more need to read next record data and continue discovery for given UUID. By returning BT_SDP_DISCOVER_UUID_CONTINUE user allows this discovery continuation.

Enums

enum [anonymous]

Helper enum to be used as return value of bt_sdp_discover_func_t. The value informs the caller to perform further pending actions or stop them.

Values:

enumerator BT_SDP_DISCOVER_UUID_STOP = 0

enumerator BT_SDP_DISCOVER_UUID_CONTINUE

enum bt_sdp_proto

Protocols to be asked about specific parameters.

Values:

enumerator BT_SDP_PROTO_RFCOMM = 0x0003

enumerator BT_SDP_PROTO_L2CAP = 0x0100

Functions

int `bt_sdp_register_service`(struct *bt_sdp_record* *service)

Register a Service Record.

Register a Service Record. Applications can make use of macros such as BT_SDP_DECLARE_SERVICE, BT_SDP_LIST, BT_SDP_SERVICE_ID, BT_SDP_SERVICE_NAME, etc. A service declaration must start with BT_SDP_NEW_SERVICE.

Parameters

- `service` – Service record declared using BT_SDP_DECLARE_SERVICE.

Returns 0 in case of success or negative value in case of error.

int `bt_sdp_discover`(struct bt_conn *conn, const struct *bt_sdp_discover_params* *params)

Allows user to start SDP discovery session.

The function performs SDP service discovery on remote server driven by user delivered discovery parameters. Discovery session is made as soon as no SDP transaction is ongoing between peers and if any then this one is queued to be processed at discovery completion of previous one. On the service discovery completion the callback function will be called to get feedback to user about findings.

Parameters

- `conn` – Object identifying connection to remote.
- `params` – SDP discovery parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_sdp_discover_cancel`(struct bt_conn *conn, const struct *bt_sdp_discover_params* *params)

Release waiting SDP discovery request.

It can cancel valid waiting SDP client request identified by SDP discovery parameters object.

Parameters

- `conn` – Object identifying connection to remote.
- `params` – SDP discovery parameters.

Returns 0 in case of success or negative value in case of error.

int `bt_sdp_get_proto_param`(const struct *net_buf* *buf, enum *bt_sdp_proto* proto, uint16_t *param)

Give to user parameter value related to given stacked protocol UUID.

API extracts specific parameter associated with given protocol UUID available in Protocol Descriptor List attribute.

Parameters

- `buf` – Original buffered raw record data.
- `proto` – Known protocol to be checked like RFCOMM or L2CAP.
- `param` – On success populated by found parameter value.

Returns 0 on success when specific parameter associated with given protocol value is found, or negative if error occurred during processing.

int `bt_sdp_get_addl_proto_param`(const struct *net_buf* *buf, enum *bt_sdp_proto* proto, uint8_t param_index, uint16_t *param)

Get additional parameter value related to given stacked protocol UUID.

API extracts specific parameter associated with given protocol UUID available in Additional Protocol Descriptor List attribute.

Parameters

- `buf` – Original buffered raw record data.
- `proto` – Known protocol to be checked like RFCOMM or L2CAP.
- `param_index` – There may be more than one parameter realted to the given protocol UUID. This function returns the result that is indexed by this parameter. It's value is from 0, 0 means the first matched result, 1 means the second matched result.
- `param` – **[out]** On success populated by found parameter value.

Returns 0 on success when a specific parameter associated with a given protocol value is found, or negative if error occurred during processing.

int `bt_sdp_get_profile_version`(const struct *net_buf* *buf, uint16_t profile, uint16_t *version)

Get profile version.

Helper API extracting remote profile version number. To get it proper generic profile parameter needs to be selected usually listed in SDP Interoperability Requirements section for given profile specification.

Parameters

- `buf` – Original buffered raw record data.
- `profile` – Profile family identifier the profile belongs.
- `version` – On success populated by found version number.

Returns 0 on success, negative value if error occurred during processing.

int `bt_sdp_get_features`(const struct *net_buf* *buf, uint16_t *features)

Get SupportedFeatures attribute value.

Allows if exposed by remote retrieve SupportedFeature attribute.

Parameters

- `buf` – Buffer holding original raw record data from remote.
- `features` – On success object to be populated with SupportedFeature mask.

Returns 0 on success if feature found and valid, negative in case any error

struct `bt_sdp_data_elem`

#include <sdp.h> SDP Generic Data Element Value.

struct `bt_sdp_attribute`

#include <sdp.h> SDP Attribute Value.

struct `bt_sdp_record`

#include <sdp.h> SDP Service Record Value.

struct `bt_sdp_client_result`

#include <sdp.h> Generic SDP Client Query Result data holder.

struct `bt_sdp_discover_params`

#include <sdp.h> Main user structure used in SDP discovery of remote.

Public Members

const struct *bt_uuid* *uuid

UUID (service) to be discovered on remote SDP entity

bt_sdp_discover_func_t func

Discover callback to be called on resolved SDP record

struct *net_buf_pool* *pool

Memory buffer enabled by user for SDP query results

7.4.14 Universal Unique Identifiers (UUIDs)

API Reference

group bt_uuid

UUIDs.

Defines

BT_UUID_SIZE_16

Size in octets of a 16-bit UUID

BT_UUID_SIZE_32

Size in octets of a 32-bit UUID

BT_UUID_SIZE_128

Size in octets of a 128-bit UUID

BT_UUID_INIT_16(value)

Initialize a 16-bit UUID.

Parameters

- value – 16-bit UUID value in host endianness.

BT_UUID_INIT_32(value)

Initialize a 32-bit UUID.

Parameters

- value – 32-bit UUID value in host endianness.

BT_UUID_INIT_128(value...)

Initialize a 128-bit UUID.

Parameters

- value – 128-bit UUID array values in little-endian format. Can be combined with *BT_UUID_128_ENCODE* to initialize a UUID from the readable form of UUIDs.

BT_UUID_DECLARE_16(value)

Helper to declare a 16-bit UUID inline.

Parameters

- value – 16-bit UUID value in host endianness.

Returns Pointer to a generic UUID.

BT_UUID_DECLARE_32(value)

Helper to declare a 32-bit UUID inline.

Parameters

- value – 32-bit UUID value in host endianness.

Returns Pointer to a generic UUID.

BT_UUID_DECLARE_128(value...)

Helper to declare a 128-bit UUID inline.

Parameters

- value – 128-bit UUID array values in little-endian format. Can be combined with *BT_UUID_128_ENCODE* to declare a UUID from the readable form of UUIDs.

Returns Pointer to a generic UUID.

BT_UUID_16(__u)

Helper macro to access the 16-bit UUID from a generic UUID.

BT_UUID_32(__u)

Helper macro to access the 32-bit UUID from a generic UUID.

BT_UUID_128(__u)

Helper macro to access the 128-bit UUID from a generic UUID.

BT_UUID_128_ENCODE(w32, w1, w2, w3, w48)

Encode 128 bit UUID into array values in little-endian format.

Helper macro to initialize a 128-bit UUID array value from the readable form of UUIDs, or encode 128-bit UUID values into advertising data Can be combined with BT_UUID_DECLARE_128 to declare a 128-bit UUID.

Example of how to declare the UUID 6E400001-B5A3-F393-E0A9-E50E24DCCA9E

```
BT_UUID_DECLARE_128(
     BT_UUID_128_ENCODE(0x6E400001, 0xB5A3, 0xF393, 0xE0A9, 0xE50E24DCCA9E))
```

Example of how to encode the UUID 6E400001-B5A3-F393-E0A9-E50E24DCCA9E into advertising data.

```
BT_DATA_BYTES(BT_DATA_UUID128_ALL,
     BT_UUID_128_ENCODE(0x6E400001, 0xB5A3, 0xF393, 0xE0A9, 0xE50E24DCCA9E))
```

Just replace the hyphen by the comma and add 0x prefixes.

Parameters

- w32 – First part of the UUID (32 bits)
- w1 – Second part of the UUID (16 bits)
- w2 – Third part of the UUID (16 bits)
- w3 – Fourth part of the UUID (16 bits)
- w48 – Fifth part of the UUID (48 bits)

Returns The comma separated values for UUID 128 initializer that may be used directly as an argument for *BT_UUID_INIT_128* or *BT_UUID_DECLARE_128*

`BT_UUID_16_ENCODE(w16)`

Encode 16-bit UUID into array values in little-endian format.

Helper macro to encode 16-bit UUID values into advertising data.

Example of how to encode the UUID `0x180a` into advertising data.

```
BT_DATA_BYTES(BT_DATA_UUID16_ALL, BT_UUID_16_ENCODE(0x180a))
```

Parameters

- `w16` – UUID value (16-bits)

Returns The comma separated values for UUID 16 value that may be used directly as an argument for *BT_DATA_BYTES*.

`BT_UUID_32_ENCODE(w32)`

Encode 32-bit UUID into array values in little-endian format.

Helper macro to encode 32-bit UUID values into advertising data.

Example of how to encode the UUID `0x180a01af` into advertising data.

```
BT_DATA_BYTES(BT_DATA_UUID32_ALL, BT_UUID_32_ENCODE(0x180a01af))
```

Parameters

- `w32` – UUID value (32-bits)

Returns The comma separated values for UUID 32 value that may be used directly as an argument for *BT_DATA_BYTES*.

`BT_UUID_STR_LEN`

Recommended length of user string buffer for Bluetooth UUID.

The recommended length guarantee the output of UUID conversion will not lose valuable information about the UUID being processed. If the length of the UUID is known the string can be shorter.

`BT_UUID_GAP_VAL`

Generic Access UUID value.

`BT_UUID_GAP`

Generic Access.

`BT_UUID_GATT_VAL`

Generic attribute UUID value.

`BT_UUID_GATT`

Generic Attribute.

`BT_UUID_IAS_VAL`

Immediate Alert Service UUID value.

BT_UUID_IAS
: Immediate Alert Service.

BT_UUID_LLS_VAL
: Link Loss Service UUID value.

BT_UUID_LLS
: Link Loss Service.

BT_UUID_TPS_VAL
: Tx Power Service UUID value.

BT_UUID_TPS
: Tx Power Service.

BT_UUID_CTS_VAL
: Current Time Service UUID value.

BT_UUID_CTS
: Current Time Service.

BT_UUID_HTS_VAL
: Health Thermometer Service UUID value.

BT_UUID_HTS
: Health Thermometer Service.

BT_UUID_DIS_VAL
: Device Information Service UUID value.

BT_UUID_DIS
: Device Information Service.

BT_UUID_HRS_VAL
: Heart Rate Service UUID value.

BT_UUID_HRS
: Heart Rate Service.

BT_UUID_BAS_VAL
: Battery Service UUID value.

BT_UUID_BAS
: Battery Service.

BT_UUID_HIDS_VAL
: HID Service UUID value.

BT_UUID_HIDS
: HID Service.

BT_UUID_RSCS_VAL
: Running Speed and Cadence Service UUID value.

BT_UUID_RSCS
: Running Speed and Cadence Service.

BT_UUID_CSC_VAL
: Cycling Speed and Cadence Service UUID value.

BT_UUID_CSC
: Cycling Speed and Cadence Service.

BT_UUID_ESS_VAL
: Environmental Sensing Service UUID value.

BT_UUID_ESS
: Environmental Sensing Service.

BT_UUID_BMS_VAL
: Bond Management Service UUID value.

BT_UUID_BMS
: Bond Management Service.

BT_UUID_IPSS_VAL
: IP Support Service UUID value.

BT_UUID_IPSS
: IP Support Service.

BT_UUID_HPS_VAL
: HTTP Proxy Service UUID value.

BT_UUID_HPS
: HTTP Proxy Service.

BT_UUID_OTS_VAL
: Object Transfer Service UUID value.

BT_UUID_OTS
: Object Transfer Service.

BT_UUID_MESH_PROV_VAL
: Mesh Provisioning Service UUID value.

BT_UUID_MESH_PROV
: Mesh Provisioning Service.

BT_UUID_MESH_PROXY_VAL
: Mesh Proxy Service UUID value.

BT_UUID_MESH_PROXY
: Mesh Proxy Service.

BT_UUID_AICS_VAL
: Audio Input Control Service value.

BT_UUID_AICS
: Audio Input Control Service.

BT_UUID_VCS_VAL
: Volume Control Service value.

BT_UUID_VCS
: Volume Control Service.

BT_UUID_VOCS_VAL
: Volume Offset Control Service value.

BT_UUID_VOCS
: Volume Offset Control Service.

BT_UUID_MICS_VAL
: Microphone Input Control Service value.

BT_UUID_MICS
: Microphone Input Control Service.

BT_UUID_GATT_PRIMARY_VAL
: GATT Primary Service UUID value.

BT_UUID_GATT_PRIMARY
: GATT Primary Service.

BT_UUID_GATT_SECONDARY_VAL
: GATT Secondary Service UUID value.

BT_UUID_GATT_SECONDARY
: GATT Secondary Service.

BT_UUID_GATT_INCLUDE_VAL
: GATT Include Service UUID value.

BT_UUID_GATT_INCLUDE
: GATT Include Service.

BT_UUID_GATT_CHRC_VAL
: GATT Characteristic UUID value.

BT_UUID_GATT_CHRC
: GATT Characteristic.

BT_UUID_GATT_CEP_VAL
: GATT Characteristic Extended Properties UUID value.

BT_UUID_GATT_CEP
: GATT Characteristic Extended Properties.

BT_UUID_GATT_CUD_VAL
: GATT Characteristic User Description UUID value.

BT_UUID_GATT_CUD
: GATT Characteristic User Description.

BT_UUID_GATT_CCC_VAL
: GATT Client Characteristic Configuration UUID value.

BT_UUID_GATT_CCC
: GATT Client Characteristic Configuration.

BT_UUID_GATT_SCC_VAL
: GATT Server Characteristic Configuration UUID value.

BT_UUID_GATT_SCC
: GATT Server Characteristic Configuration.

BT_UUID_GATT_CPF_VAL
: GATT Characteristic Presentation Format UUID value.

BT_UUID_GATT_CPF
: GATT Characteristic Presentation Format.

BT_UUID_GATT_CAF_VAL
: GATT Characteristic Aggregated Format UUID value.

BT_UUID_GATT_CAF
: GATT Characteristic Aggregated Format.

BT_UUID_VALID_RANGE_VAL
: Valid Range Descriptor UUID value.

BT_UUID_VALID_RANGE
: Valid Range Descriptor.

BT_UUID_HIDS_EXT_REPORT_VAL
: HID External Report Descriptor UUID value.

BT_UUID_HIDS_EXT_REPORT
: HID External Report Descriptor.

BT_UUID_HIDS_REPORT_REF_VAL
: HID Report Reference Descriptor UUID value.

BT_UUID_HIDS_REPORT_REF
: HID Report Reference Descriptor.

BT_UUID_ES_CONFIGURATION_VAL
: Environmental Sensing Configuration Descriptor UUID value.

BT_UUID_ES_CONFIGURATION
: Environmental Sensing Configuration Descriptor.

BT_UUID_ES_MEASUREMENT_VAL
: Environmental Sensing Measurement Descriptor UUID value.

BT_UUID_ES_MEASUREMENT
: Environmental Sensing Measurement Descriptor.

BT_UUID_ES_TRIGGER_SETTING_VAL
: Environmental Sensing Trigger Setting Descriptor UUID value.

BT_UUID_ES_TRIGGER_SETTING
: Environmental Sensing Trigger Setting Descriptor.

BT_UUID_GAP_DEVICE_NAME_VAL
: GAP Characteristic Device Name UUID value.

BT_UUID_GAP_DEVICE_NAME
: GAP Characteristic Device Name.

BT_UUID_GAP_APPEARANCE_VAL
: GAP Characteristic Appearance UUID value.

BT_UUID_GAP_APPEARANCE
: GAP Characteristic Appearance.

BT_UUID_GAP_PPCP_VAL
: GAP Characteristic Peripheral Preferred Connection Parameters UUID value.

BT_UUID_GAP_PPCP
: GAP Characteristic Peripheral Preferred Connection Parameters.

BT_UUID_GATT_SC_VAL
: GATT Characteristic Service Changed UUID value.

BT_UUID_GATT_SC
: GATT Characteristic Service Changed.

BT_UUID_ALERT_LEVEL_VAL
: Alert Level UUID value.

BT_UUID_ALERT_LEVEL
: Alert Level.

BT_UUID_TPS_TX_POWER_LEVEL_VAL
: TPS Characteristic Tx Power Level UUID value.

BT_UUID_TPS_TX_POWER_LEVEL
: TPS Characteristic Tx Power Level.

BT_UUID_BAS_BATTERY_LEVEL_VAL
: BAS Characteristic Battery Level UUID value.

BT_UUID_BAS_BATTERY_LEVEL
: BAS Characteristic Battery Level.

BT_UUID_HTS_MEASUREMENT_VAL
: HTS Characteristic Measurement Value UUID value.

BT_UUID_HTS_MEASUREMENT
: HTS Characteristic Measurement Value.

BT_UUID_HIDS_BOOT_KB_IN_REPORT_VAL
: HID Characteristic Boot Keyboard Input Report UUID value.

BT_UUID_HIDS_BOOT_KB_IN_REPORT
: HID Characteristic Boot Keyboard Input Report.

BT_UUID_DIS_SYSTEM_ID_VAL
: DIS Characteristic System ID UUID value.

BT_UUID_DIS_SYSTEM_ID
: DIS Characteristic System ID.

BT_UUID_DIS_MODEL_NUMBER_VAL
: DIS Characteristic Model Number String UUID value.

BT_UUID_DIS_MODEL_NUMBER
: DIS Characteristic Model Number String.

BT_UUID_DIS_SERIAL_NUMBER_VAL
: DIS Characteristic Serial Number String UUID value.

BT_UUID_DIS_SERIAL_NUMBER
: DIS Characteristic Serial Number String.

BT_UUID_DIS_FIRMWARE_REVISION_VAL
: DIS Characteristic Firmware Revision String UUID value.

BT_UUID_DIS_FIRMWARE_REVISION
: DIS Characteristic Firmware Revision String.

BT_UUID_DIS_HARDWARE_REVISION_VAL
: DIS Characteristic Hardware Revision String UUID value.

BT_UUID_DIS_HARDWARE_REVISION
: DIS Characteristic Hardware Revision String.

BT_UUID_DIS_SOFTWARE_REVISION_VAL
: DIS Characteristic Software Revision String UUID value.

BT_UUID_DIS_SOFTWARE_REVISION
: DIS Characteristic Software Revision String.

BT_UUID_DIS_MANUFACTURER_NAME_VAL
: DIS Characteristic Manufacturer Name String UUID Value.

BT_UUID_DIS_MANUFACTURER_NAME
: DIS Characteristic Manufacturer Name String.

BT_UUID_DIS_PNP_ID_VAL
: DIS Characteristic PnP ID UUID value.

BT_UUID_DIS_PNP_ID
: DIS Characteristic PnP ID.

BT_UUID_CTS_CURRENT_TIME_VAL
: CTS Characteristic Current Time UUID value.

BT_UUID_CTS_CURRENT_TIME
: CTS Characteristic Current Time.

BT_UUID_MAGN_DECLINATION_VAL
: Magnetic Declination Characteristic UUID value.

BT_UUID_MAGN_DECLINATION
: Magnetic Declination Characteristic.

BT_UUID_HIDS_BOOT_KB_OUT_REPORT_VAL
: HID Boot Keyboard Output Report Characteristic UUID value.

BT_UUID_HIDS_BOOT_KB_OUT_REPORT
: HID Boot Keyboard Output Report Characteristic.

BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT_VAL
: HID Boot Mouse Input Report Characteristic UUID value.

BT_UUID_HIDS_BOOT_MOUSE_IN_REPORT
: HID Boot Mouse Input Report Characteristic.

BT_UUID_HRS_MEASUREMENT_VAL
: HRS Characteristic Measurement Interval UUID value.

BT_UUID_HRS_MEASUREMENT
: HRS Characteristic Measurement Interval.

BT_UUID_HRS_BODY_SENSOR
: HRS Characteristic Body Sensor Location.

BT_UUID_HRS_BODY_SENSOR_VAL

BT_UUID_HRS_CONTROL_POINT
: HRS Characteristic Control Point.

BT_UUID_HRS_CONTROL_POINT_VAL
: HRS Characteristic Control Point UUID value.

BT_UUID_HIDS_INFO_VAL
: HID Information Characteristic UUID value.

BT_UUID_HIDS_INFO
: HID Information Characteristic.

BT_UUID_HIDS_REPORT_MAP_VAL
: HID Report Map Characteristic UUID value.

BT_UUID_HIDS_REPORT_MAP
: HID Report Map Characteristic.

BT_UUID_HIDS_CTRL_POINT_VAL
: HID Control Point Characteristic UUID value.

BT_UUID_HIDS_CTRL_POINT
: HID Control Point Characteristic.

BT_UUID_HIDS_REPORT_VAL
: HID Report Characteristic UUID value.

BT_UUID_HIDS_REPORT
: HID Report Characteristic.

BT_UUID_HIDS_PROTOCOL_MODE_VAL
: HID Protocol Mode Characteristic UUID value.

BT_UUID_HIDS_PROTOCOL_MODE
: HID Protocol Mode Characteristic.

BT_UUID_RSC_MEASUREMENT_VAL
: RSC Measurement Characteristic UUID value.

BT_UUID_RSC_MEASUREMENT
: RSC Measurement Characteristic.

BT_UUID_RSC_FEATURE_VAL
: RSC Feature Characteristic UUID value.

BT_UUID_RSC_FEATURE
: RSC Feature Characteristic.

BT_UUID_CSC_MEASUREMENT_VAL
: CSC Measurement Characteristic UUID value.

BT_UUID_CSC_MEASUREMENT
: CSC Measurement Characteristic.

BT_UUID_CSC_FEATURE_VAL
: CSC Feature Characteristic UUID value.

BT_UUID_CSC_FEATURE
: CSC Feature Characteristic.

BT_UUID_SENSOR_LOCATION_VAL
: Sensor Location Characteristic UUID value.

BT_UUID_SENSOR_LOCATION
: Sensor Location Characteristic.

BT_UUID_SC_CONTROL_POINT_VAL
: SC Control Point Characteristic UUID value.

BT_UUID_SC_CONTROL_POINT
: SC Control Point Characteristic.

BT_UUID_ELEVATION_VAL
: Elevation Characteristic UUID value.

BT_UUID_ELEVATION
: Elevation Characteristic.

BT_UUID_PRESSURE_VAL
: Pressure Characteristic UUID value.

BT_UUID_PRESSURE
: Pressure Characteristic.

BT_UUID_TEMPERATURE_VAL
: Temperature Characteristic UUID value.

BT_UUID_TEMPERATURE
: Temperature Characteristic.

BT_UUID_HUMIDITY_VAL
: Humidity Characteristic UUID value.

BT_UUID_HUMIDITY
: Humidity Characteristic.

BT_UUID_TRUE_WIND_SPEED_VAL
: True Wind Speed Characteristic UUID value.

BT_UUID_TRUE_WIND_SPEED
: True Wind Speed Characteristic.

BT_UUID_TRUE_WIND_DIR_VAL
: True Wind Direction Characteristic UUID value.

BT_UUID_TRUE_WIND_DIR
: True Wind Direction Characteristic.

BT_UUID_APPARENT_WIND_SPEED_VAL
: Apparent Wind Speed Characteristic UUID value.

BT_UUID_APPARENT_WIND_SPEED
: Apparent Wind Speed Characteristic.

BT_UUID_APPARENT_WIND_DIR_VAL
: Apparent Wind Direction Characteristic UUID value.

BT_UUID_APPARENT_WIND_DIR
: Apparent Wind Direction Characteristic.

BT_UUID_GUST_FACTOR_VAL
: Gust Factor Characteristic UUID value.

BT_UUID_GUST_FACTOR
: Gust Factor Characteristic.

BT_UUID_POLLEN_CONCENTRATION_VAL
: Pollen Concentration Characteristic UUID value.

BT_UUID_POLLEN_CONCENTRATION
: Pollen Concentration Characteristic.

BT_UUID_UV_INDEX_VAL
: UV Index Characteristic UUID value.

BT_UUID_UV_INDEX
: UV Index Characteristic.

BT_UUID_IRRADIANCE_VAL
: Irradiance Characteristic UUID value.

BT_UUID_IRRADIANCE
: Irradiance Characteristic.

BT_UUID_RAINFALL_VAL
: Rainfall Characteristic UUID value.

BT_UUID_RAINFALL
: Rainfall Characteristic.

BT_UUID_WIND_CHILL_VAL
: Wind Chill Characteristic UUID value.

BT_UUID_WIND_CHILL
: Wind Chill Characteristic.

BT_UUID_HEAT_INDEX_VAL
: Heat Index Characteristic UUID value.

BT_UUID_HEAT_INDEX
: Heat Index Characteristic.

BT_UUID_DEW_POINT_VAL
: Dew Point Characteristic UUID value.

BT_UUID_DEW_POINT
: Dew Point Characteristic.

BT_UUID_DESC_VALUE_CHANGED_VAL
: Descriptor Value Changed Characteristic UUID value.

BT_UUID_DESC_VALUE_CHANGED
: Descriptor Value Changed Characteristic.

BT_UUID_MAGN_FLUX_DENSITY_2D_VAL
: Magnetic Flux Density - 2D Characteristic UUID value.

BT_UUID_MAGN_FLUX_DENSITY_2D
: Magnetic Flux Density - 2D Characteristic.

BT_UUID_MAGN_FLUX_DENSITY_3D_VAL
: Magnetic Flux Density - 3D Characteristic UUID value.

BT_UUID_MAGN_FLUX_DENSITY_3D
: Magnetic Flux Density - 3D Characteristic.

BT_UUID_BAR_PRESSURE_TREND_VAL
: Barometric Pressure Trend Characteristic UUID value.

BT_UUID_BAR_PRESSURE_TREND
: Barometric Pressure Trend Characteristic.

BT_UUID_BMS_CONTROL_POINT_VAL
: Bond Management Control Point UUID value.

BT_UUID_BMS_CONTROL_POINT
: Bond Management Control Point.

BT_UUID_BMS_FEATURE_VAL
: Bond Management Feature UUID value.

BT_UUID_BMS_FEATURE
: Bond Management Feature.

BT_UUID_CENTRAL_ADDR_RES_VAL
: Central Address Resolution Characteristic UUID value.

BT_UUID_CENTRAL_ADDR_RES
: Central Address Resolution Characteristic.

BT_UUID_URI_VAL
: URI UUID value.

BT_UUID_URI
: URI.

BT_UUID_HTTP_HEADERS_VAL
: HTTP Headers UUID value.

BT_UUID_HTTP_HEADERS
: HTTP Headers.

BT_UUID_HTTP_STATUS_CODE_VAL
: HTTP Status Code UUID value.

BT_UUID_HTTP_STATUS_CODE
: HTTP Status Code.

BT_UUID_HTTP_ENTITY_BODY_VAL
: HTTP Entity Body UUID value.

BT_UUID_HTTP_ENTITY_BODY
: HTTP Entity Body.

BT_UUID_HTTP_CONTROL_POINT_VAL
: HTTP Control Point UUID value.

BT_UUID_HTTP_CONTROL_POINT
: HTTP Control Point.

BT_UUID_HTTPS_SECURITY_VAL
: HTTPS Security UUID value.

BT_UUID_HTTPS_SECURITY
: HTTPS Security.

BT_UUID_OTS_FEATURE_VAL
: OTS Feature Characteristic UUID value.

BT_UUID_OTS_FEATURE
: OTS Feature Characteristic.

BT_UUID_OTS_NAME_VAL
: OTS Object Name Characteristic UUID value.

BT_UUID_OTS_NAME
: OTS Object Name Characteristic.

BT_UUID_OTS_TYPE_VAL
: OTS Object Type Characteristic UUID value.

BT_UUID_OTS_TYPE
: OTS Object Type Characteristic.

BT_UUID_OTS_SIZE_VAL
: OTS Object Size Characteristic UUID value.

BT_UUID_OTS_SIZE
: OTS Object Size Characteristic.

BT_UUID_OTS_FIRST_CREATED_VAL
: OTS Object First-Created Characteristic UUID value.

BT_UUID_OTS_FIRST_CREATED
: OTS Object First-Created Characteristic.

BT_UUID_OTS_LAST_MODIFIED_VAL
: OTS Object Last-Modified Characteristic UUI value.

BT_UUID_OTS_LAST_MODIFIED
: OTS Object Last-Modified Characteristic.

BT_UUID_OTS_ID_VAL
: OTS Object ID Characteristic UUID value.

BT_UUID_OTS_ID
: OTS Object ID Characteristic.

BT_UUID_OTS_PROPERTIES_VAL
: OTS Object Properties Characteristic UUID value.

BT_UUID_OTS_PROPERTIES
: OTS Object Properties Characteristic.

BT_UUID_OTS_ACTION_CP_VAL
: OTS Object Action Control Point Characteristic UUID value.

BT_UUID_OTS_ACTION_CP
: OTS Object Action Control Point Characteristic.

BT_UUID_OTS_LIST_CP_VAL
: OTS Object List Control Point Characteristic UUID value.

BT_UUID_OTS_LIST_CP
: OTS Object List Control Point Characteristic.

BT_UUID_OTS_LIST_FILTER_VAL
: OTS Object List Filter Characteristic UUID value.

BT_UUID_OTS_LIST_FILTER
: OTS Object List Filter Characteristic.

BT_UUID_OTS_CHANGED_VAL
: OTS Object Changed Characteristic UUID value.

BT_UUID_OTS_CHANGED
: OTS Object Changed Characteristic.

BT_UUID_OTS_TYPE_UNSPECIFIED_VAL
: OTS Unspecified Object Type UUID value.

BT_UUID_OTS_TYPE_UNSPECIFIED
: OTS Unspecified Object Type.

BT_UUID_OTS_DIRECTORY_LISTING_VAL
: OTS Directory Listing UUID value.

BT_UUID_OTS_DIRECTORY_LISTING
: OTS Directory Listing.

BT_UUID_MESH_PROV_DATA_IN_VAL
: Mesh Provisioning Data In UUID value.

BT_UUID_MESH_PROV_DATA_IN
: Mesh Provisioning Data In.

BT_UUID_MESH_PROV_DATA_OUT_VAL
: Mesh Provisioning Data Out UUID value.

BT_UUID_MESH_PROV_DATA_OUT
: Mesh Provisioning Data Out.

BT_UUID_MESH_PROXY_DATA_IN_VAL
: Mesh Proxy Data In UUID value.

BT_UUID_MESH_PROXY_DATA_IN
: Mesh Proxy Data In.

BT_UUID_MESH_PROXY_DATA_OUT_VAL
: Mesh Proxy Data Out UUID value.

BT_UUID_MESH_PROXY_DATA_OUT
: Mesh Proxy Data Out.

BT_UUID_GATT_CLIENT_FEATURES_VAL
: Client Supported Features UUID value.

BT_UUID_GATT_CLIENT_FEATURES
: Client Supported Features.

BT_UUID_GATT_DB_HASH_VAL
: Database Hash UUID value.

BT_UUID_GATT_DB_HASH
: Database Hash.

BT_UUID_GATT_SERVER_FEATURES_VAL
: Server Supported Features UUID value.

BT_UUID_GATT_SERVER_FEATURES
: Server Supported Features.

BT_UUID_AICS_STATE_VAL
: Audio Input Control Service State value.

BT_UUID_AICS_STATE
: Audio Input Control Service State.

BT_UUID_AICS_GAIN_SETTINGS_VAL
: Audio Input Control Service Gain Settings Properties value.

BT_UUID_AICS_GAIN_SETTINGS
: Audio Input Control Service Gain Settings Properties.

BT_UUID_AICS_INPUT_TYPE_VAL
: Audio Input Control Service Input Type value.

BT_UUID_AICS_INPUT_TYPE
: Audio Input Control Service Input Type.

BT_UUID_AICS_INPUT_STATUS_VAL
: Audio Input Control Service Input Status value.

BT_UUID_AICS_INPUT_STATUS
: Audio Input Control Service Input Status.

BT_UUID_AICS_CONTROL_VAL
: Audio Input Control Service Control Point value.

BT_UUID_AICS_CONTROL
: Audio Input Control Service Control Point.

BT_UUID_AICS_DESCRIPTION_VAL
: Audio Input Control Service Input Description value.

BT_UUID_AICS_DESCRIPTION
: Audio Input Control Service Input Description.

BT_UUID_VCS_STATE_VAL
: Volume Control Setting value.

BT_UUID_VCS_STATE
: Volume Control Setting.

BT_UUID_VCS_CONTROL_VAL
: Volume Control Control point value.

BT_UUID_VCS_CONTROL
: Volume Control Control point.

BT_UUID_VCS_FLAGS_VAL
: Volume Control Flags value.

BT_UUID_VCS_FLAGS
: Volume Control Flags.

BT_UUID_VOCS_STATE_VAL
: Volume Offset State value.

BT_UUID_VOCS_STATE
: Volume Offset State.

BT_UUID_VOCS_LOCATION_VAL
: Audio Location value.

BT_UUID_VOCS_LOCATION
: Audio Location.

BT_UUID_VOCS_CONTROL_VAL
: Volume Offset Control Point value.

BT_UUID_VOCS_CONTROL
: Volume Offset Control Point.

BT_UUID_VOCS_DESCRIPTION_VAL
: Volume Offset Audio Output Description value.

BT_UUID_VOCS_DESCRIPTION
: Volume Offset Audio Output Description.

BT_UUID_MICS_MUTE_VAL
: Microphone Input Control Service Mute value.

BT_UUID_MICS_MUTE
: Microphone Input Control Service Mute.

BT_UUID_SDP_VAL

BT_UUID_SDP

BT_UUID_UDP_VAL

BT_UUID_UDP

BT_UUID_RFCOMM_VAL

BT_UUID_RFCOMM

BT_UUID_TCP_VAL

BT_UUID_TCP

BT_UUID_TCS_BIN_VAL

BT_UUID_TCS_BIN

BT_UUID_TCS_AT_VAL

BT_UUID_TCS_AT

BT_UUID_ATT_VAL

BT_UUID_ATT

BT_UUID_OBEX_VAL

BT_UUID_OBEX

BT_UUID_IP_VAL

BT_UUID_IP

BT_UUID_FTP_VAL

BT_UUID_FTP

BT_UUID_HTTP_VAL

BT_UUID_HTTP

BT_UUID_BNEP_VAL

BT_UUID_BNEP

BT_UUID_UPNP_VAL

BT_UUID_UPNP

BT_UUID_HIDP_VAL

BT_UUID_HIDP

BT_UUID_HCRP_CTRL_VAL

BT_UUID_HCRP_CTRL

BT_UUID_HCRP_DATA_VAL

BT_UUID_HCRP_DATA

BT_UUID_HCRP_NOTE_VAL

BT_UUID_HCRP_NOTE

BT_UUID_AVCTP_VAL

BT_UUID_AVCTP

BT_UUID_AVDTP_VAL

BT_UUID_AVDTP

BT_UUID_CMTP_VAL

BT_UUID_CMTP

BT_UUID_UDI_VAL

BT_UUID_UDI

BT_UUID_MCAP_CTRL_VAL

BT_UUID_MCAP_CTRL

BT_UUID_MCAP_DATA_VAL

BT_UUID_MCAP_DATA

BT_UUID_L2CAP_VAL

BT_UUID_L2CAP

Enums

enum [anonymous]

Bluetooth UUID types.

Values:

enumerator BT_UUID_TYPE_16

UUID type 16-bit.

enumerator BT_UUID_TYPE_32

UUID type 32-bit.

enumerator BT_UUID_TYPE_128

UUID type 128-bit.

Functions

int bt_uuid_cmp(const struct *bt_uuid* *u1, const struct *bt_uuid* *u2)

Compare Bluetooth UUIDs.

Compares 2 Bluetooth UUIDs, if the types are different both UUIDs are first converted to 128 bits format before comparing.

Parameters

- u1 – First Bluetooth UUID to compare
- u2 – Second Bluetooth UUID to compare

Returns negative value if $u1 < u2$, 0 if $u1 == u2$, else positive

bool bt_uuid_create(struct *bt_uuid* *uuid, const uint8_t *data, uint8_t data_len)

Create a *bt_uuid* from a little-endian data buffer.

Create a *bt_uuid* from a little-endian data buffer. The data_len parameter is used to determine whether the UUID is in 16, 32 or 128 bit format (length 2, 4 or 16). Note: 32 bit format is not allowed over the air.

Parameters

- uuid – Pointer to the *bt_uuid* variable
- data – pointer to UUID stored in little-endian data buffer
- data_len – length of the UUID in the data buffer

Returns true if the data was valid and the UUID was successfully created.

void bt_uuid_to_str(const struct *bt_uuid* *uuid, char *str, size_t len)

Convert Bluetooth UUID to string.

Converts Bluetooth UUID to string. UUID can be in any format, 16-bit, 32-bit or 128-bit.

Parameters

- `uuid` – Bluetooth UUID
- `str` – pointer where to put converted string
- `len` – length of str

Returns N/A

struct `bt_uuid`

#include <uuid.h> This is a 'tentative' type and should be used as a pointer only.

struct `bt_uuid_16`

#include <uuid.h>

Public Members

struct *bt_uuid* `uuid`

UUID generic type.

uint16_t `val`

UUID value, 16-bit in host endianness.

struct `bt_uuid_32`

#include <uuid.h>

Public Members

struct *bt_uuid* `uuid`

UUID generic type.

uint32_t `val`

UUID value, 32-bit in host endianness.

struct `bt_uuid_128`

#include <uuid.h>

Public Members

struct *bt_uuid* `uuid`

UUID generic type.

uint8_t `val`[16]

UUID value, 128-bit in little-endian format.

7.5 Crypto

7.5.1 Overview

7.5.2 API Reference

group `crypto_cipher`

Crypto Cipher APIs.

Defines

`CAP_OPAQUE_KEY_HNDL`

`CAP_RAW_KEY`

`CAP_KEY_LOADING_API`

`CAP_INPLACE_OPS`

Whether the output is placed in separate buffer or not

`CAP_SEPARATE_IO_BUFS`

`CAP_SYNC_OPS`

These denotes if the output (completion of a cipher_xxx_op) is conveyed by the op function returning, or it is conveyed by an async notification

`CAP_ASYNC_OPS`

`CAP_AUTONONCE`

Whether the hardware/driver supports autononce feature

`CAP_NO_IV_PREFIX`

Don't prefix IV to cipher blocks

Typedefs

typedef int (*`block_op_t`)(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt)

typedef int (*`cbc_op_t`)(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *iv)

typedef int (*`ctr_op_t`)(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *ctr)

typedef int (*`ccm_op_t`)(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

typedef int (*`gcm_op_t`)(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

typedef void (*`crypto_completion_cb`)(struct *cipher_pkt* *completed, int status)

Enums

enum `cipher_algo`

Cipher Algorithm

Values:

enumerator `CRYPTO_CIPHER_ALGO_AES` = 1

enum `cipher_op`

Cipher Operation

Values:

enumerator `CRYPTO_CIPHER_OP_DECRYPT` = 0

enumerator `CRYPTO_CIPHER_OP_ENCRYPT` = 1

enum `cipher_mode`

Possible cipher mode options.

More to be added as required.

Values:

enumerator `CRYPTO_CIPHER_MODE_ECB` = 1

enumerator `CRYPTO_CIPHER_MODE_CBC` = 2

enumerator `CRYPTO_CIPHER_MODE_CTR` = 3

enumerator `CRYPTO_CIPHER_MODE_CCM` = 4

enumerator `CRYPTO_CIPHER_MODE_GCM` = 5

Functions

static inline int `cipher_query_hwcaps`(const struct *device* *dev)

Query the crypto hardware capabilities.

This API is used by the app to query the capabilities supported by the crypto device. Based on this the app can specify a subset of the supported options to be honored for a session during *cipher_begin_session()*.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Returns bitmask of supported options.

static inline int `cipher_begin_session`(const struct *device* *dev, struct *cipher_ctx* *ctx, enum *cipher_algo* algo, enum *cipher_mode* mode, enum *cipher_op* optype)

Setup a crypto session.

Initializes one time parameters, like the session key, algorithm and cipher mode which may remain constant for all operations in the session. The state may be cached in hardware and/or driver data state variables.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ctx` – Pointer to the context structure. Various one time parameters like key, keylength, etc. are supplied via this structure. The structure documentation specifies which fields are to be populated by the app before making this call.
- `algo` – The crypto algorithm to be used in this session. e.g AES
- `mode` – The cipher mode to be used in this session. e.g CBC, CTR
- `optype` – Whether we should encrypt or decrypt in this session

Returns 0 on success, negative errno code on fail.

static inline int `cipher_free_session`(const struct *device* *dev, struct *cipher_ctx* *ctx)

Cleanup a crypto session.

Clears the hardware and/or driver state of a previous session.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ctx` – Pointer to the crypto context structure of the session to be freed.

Returns 0 on success, negative errno code on fail.

static inline int `cipher_callback_set`(const struct *device* *dev, *crypto_completion_cb* cb)

Registers an async crypto op completion callback with the driver.

The application can register an async crypto op completion callback handler to be invoked by the driver, on completion of a prior request submitted via crypto_do_op(). Based on crypto device hardware semantics, this is likely to be invoked from an ISR context.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cb` – Pointer to application callback to be called by the driver.

Returns 0 on success, -ENOTSUP if the driver does not support async op, negative errno code on other error.

static inline int `cipher_block_op`(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt)

Perform single-block crypto operation (ECB cipher mode). This should not be overloaded to operate on multiple blocks for security reasons.

Parameters

- `ctx` – Pointer to the crypto context of this op.
- `pkt` – Structure holding the input/output buffer pointers.

Returns 0 on success, negative errno code on fail.

static inline int `cipher_cbc_op`(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *iv)

Perform Cipher Block Chaining (CBC) crypto operation.

Parameters

- `ctx` – Pointer to the crypto context of this op.
- `pkt` – Structure holding the input/output buffer pointers.

- `iv` – Initialization Vector (IV) for the operation. Same IV value should not be reused across multiple operations (within a session context) for security.

Returns 0 on success, negative errno code on fail.

static inline int `cipher_ctr_op`(struct *cipher_ctx* *ctx, struct *cipher_pkt* *pkt, uint8_t *iv)

Perform Counter (CTR) mode crypto operation.

Parameters

- `ctx` – Pointer to the crypto context of this op.
- `pkt` – Structure holding the input/output buffer pointers.
- `iv` – Initialization Vector (IV) for the operation. We use a split counter formed by appending IV and ctr. Consequently ivlen = keylen - ctrlen. 'ctrlen' is specified during session setup through the 'ctx.mode_params.ctr_params.ctr_len' parameter. IV should not be reused across multiple operations (within a session context) for security. The non-IV part of the split counter is transparent to the caller and is fully managed by the crypto provider.

Returns 0 on success, negative errno code on fail.

static inline int `cipher_ccm_op`(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

Perform Counter with CBC-MAC (CCM) mode crypto operation.

Parameters

- `ctx` – Pointer to the crypto context of this op.
- `pkt` – Structure holding the input/output, Assosciated Data (AD) and auth tag buffer pointers.
- `nonce` – Nonce for the operation. Same nonce value should not be reused across multiple operations (within a session context) for security.

Returns 0 on success, negative errno code on fail.

static inline int `cipher_gcm_op`(struct *cipher_ctx* *ctx, struct *cipher_aead_pkt* *pkt, uint8_t *nonce)

Perform Galois/Counter Mode (GCM) crypto operation.

Parameters

- `ctx` – Pointer to the crypto context of this op.
- `pkt` – Structure holding the input/output, Associated Data (AD) and auth tag buffer pointers.
- `nonce` – Nonce for the operation. Same nonce value should not be reused across multiple operations (within a session context) for security.

Returns 0 on success, negative errno code on fail.

struct `crypto_driver_api`

#include <cipher.h>

struct `cipher_ops`

#include <cipher_structs.h>

struct `ccm_params`

#include <cipher_structs.h>

struct `ctr_params`

#include <cipher_structs.h>

struct `gcm_params`

#include <cipher_structs.h>

struct `cipher_ctx`

#include <cipher_structs.h> Structure encoding session parameters.

Refer to comments for individual fields to know the contract in terms of who fills what and when w.r.t begin_session() call.

Public Members

struct *cipher_ops* `ops`

Place for driver to return function pointers to be invoked per cipher operation. To be populated by crypto driver on return from begin_session() based on the algo/mode chosen by the app.

union *cipher_ctx*.[anonymous] `key`

To be populated by the app before calling begin_session()

const struct *device* `*device`

The device driver instance this crypto context relates to. Will be populated by the begin_session() API.

void `*drv_sessn_state`

If the driver supports multiple simultaneously crypto sessions, this will identify the specific driver state this crypto session relates to. Since dynamic memory allocation is not possible, it is suggested that at build time drivers allocate space for the max simultaneous sessions they intend to support. To be populated by the driver on return from begin_session().

void `*app_sessn_state`

Place for the user app to put info relevant stuff for resuming when completion callback happens for async ops. Totally managed by the app.

union *cipher_ctx*.[anonymous] `mode_params`

Cypher mode parameters, which remain constant for all ops in a session. To be populated by the app before calling begin_session().

uint16_t `keylen`

Cryptographic keylength in bytes. To be populated by the app before calling begin_session()

uint16_t `flags`

How certain fields are to be interpreted for this session. (A bitmask of CAP_* below.) To be populated by the app before calling begin_session(). An app can obtain the capability flags supported by a hw/driver by calling *cipher_query_hwcaps()*.

struct `cipher_pkt`

#include <cipher_structs.h> Structure encoding IO parameters of one cryptographic operation like encrypt/decrypt.

The fields which has not been explicitly called out has to be filled up by the app before making the cipher_xxx_op() call.

Public Members

uint8_t *`in_buf`

Start address of input buffer

int `in_len`

Bytes to be operated upon

uint8_t *`out_buf`

Start of the output buffer, to be allocated by the application. Can be NULL for in-place ops. To be populated with contents by the driver on return from op / async callback.

int `out_buf_max`

Size of the out_buf area allocated by the application. Drivers should not write past the size of output buffer.

int `out_len`

To be populated by driver on return from cipher_xxx_op() and holds the size of the actual result.

struct *cipher_ctx* *`ctx`

Context this packet relates to. This can be useful to get the session details, especially for async ops. Will be populated by the cipher_xxx_op() API based on the ctx parameter.

struct `cipher_aead_pkt`

#include <cipher_structs.h> Structure encoding IO parameters in AEAD (Authenticated Encryption with Associated Data) scenario like in CCM.

App has to furnish valid contents prior to making *cipher_ccm_op()* call.

Public Members

uint8_t *`ad`

Start address for Associated Data. This has to be supplied by app.

uint32_t `ad_len`

Size of Associated Data. This has to be supplied by the app.

uint8_t *`tag`

Start address for the auth hash. For an encryption op this will be populated by the driver when it returns from cipher_ccm_op call. For a decryption op this has to be supplied by the app.

7.6 Devicetree

This is reference documentation for devicetree as it is used for Zephyr development. For a high-level guide, see *Devicetree Guide*. For a platform-independent specification, see the Devicetree specification.

7.6.1 Devicetree API

This is a reference page for the `<devicetree.h>` API. The API is macro based. Use of these macros has no impact on scheduling. They can be used from any calling context and at file scope.

Some of these require a special macro named `DT_DRV_COMPAT` to be defined before they can be used; these are discussed individually below. These macros are generally meant for use within *device drivers*, though they can be used outside of drivers with appropriate care.

Generic APIs

The APIs in this section can be used anywhere and do not require `DT_DRV_COMPAT` to be defined.

Node identifiers and helpers A *node identifier* is a way to refer to a devicetree node at C preprocessor time. While node identifiers are not C values, you can use them to access devicetree data in C rvalue form using, for example, the *Property access* API.

The root node `/` has node identifier `DT_ROOT`. You can create node identifiers for other devicetree nodes using *`DT_PATH()`*, *`DT_NODELABEL()`*, *`DT_ALIAS()`*, and *`DT_INST()`*.

There are also *`DT_PARENT()`* and *`DT_CHILD()`* macros which can be used to create node identifiers for a given node's parent node or a particular child node, respectively.

The following macros create or operate on node identifiers.

group `devicetree-generic-id`

Defines

`DT_INVALID_NODE`

Name for an invalid node identifier.

This supports cases where factored macros can be invoked from paths where devicetree data may or may not be available. It is a preprocessor identifier that does not match any valid devicetree node identifier.

`DT_ROOT`

Node identifier for the root node in the devicetree.

`DT_PATH(...)`

Get a node identifier for a devicetree path.

(This macro returns a node identifier from path components. To get a path string from a node identifier, use *`DT_NODE_PATH()`* instead.)

The arguments to this macro are the names of non-root nodes in the tree required to reach the desired node, starting from the root. Non-alphanumeric characters in each name must be converted to underscores to form valid C tokens, and letters must be lowercased.

Example devicetree fragment:

```
/ {
        soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                        ...
                };
        };
};
```

You can use *DT_PATH(soc, serial_40001000)* to get a node identifier for the serial@40001000 node. Node labels like "serial1" cannot be used as *DT_PATH()* arguments; use *DT_NODELABEL()* for those instead.

Example usage with *DT_PROP()* to get the current-speed property:

```
DT_PROP(DT_PATH(soc, serial_40001000), current_speed) // 115200
```

(The current-speed property is also in "lowercase-and-underscores" form when used with this API.)

When determining arguments to *DT_PATH()*:

- the first argument corresponds to a child node of the root ("soc" above)
- a second argument corresponds to a child of the first argument ("serial_40001000" above, from the node name "serial@40001000" after lowercasing and changing "@" to "_")
- and so on for deeper nodes in the desired node's path

Parameters

- ... – lowercase-and-underscores node names along the node's path, with each name given as a separate argument

Returns node identifier for the node with that path

`DT_NODELABEL(label)`

Get a node identifier for a node label.

Convert non-alphanumeric characters in the node label to underscores to form valid C tokens, and lowercase all letters. Note that node labels are not the same thing as label properties.

Example devicetree fragment:

```
serial1: serial@40001000 {
        label = "UART_0";
        status = "okay";
        current-speed = <115200>;
        ...
};
```

The only node label in this example is "serial1".

The string "UART_0" is *not* a node label; it's the value of a property named label.

You can use *DT_NODELABEL(serial1)* to get a node identifier for the serial@40001000 node. Example usage with *DT_PROP()* to get the current-speed property:

```
DT_PROP(DT_NODELABEL(serial1), current_speed) // 115200
```

Another example devicetree fragment:

```
cpu@0 {
       L2_0: l2-cache {
               cache-level = <2>;
               ...
       };
};
```

Example usage to get the cache-level property:

```
DT_PROP(DT_NODELABEL(l2_0), cache_level) // 2
```

Notice how "L2_0" in the devicetree is lowercased to "l2_0" in the *DT_NODELABEL()* argument.

Parameters

- `label` – lowercase-and-underscores node label name

Returns node identifier for the node with that label

`DT_ALIAS(alias)`

Get a node identifier from /aliases.

This macro's argument is a property of the /aliases node. It returns a node identifier for the node which is aliased. Convert non-alphanumeric characters in the alias property to underscores to form valid C tokens, and lowercase all letters.

Example devicetree fragment:

```
/ {
        aliases {
                my-serial = &serial1;
        };

        soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                        ...
                };
        };
};
```

You can use *DT_ALIAS(my_serial)* to get a node identifier for the serial@40001000 node. Notice how my-serial in the devicetree becomes my_serial in the *DT_ALIAS()* argument. Example usage with *DT_PROP()* to get the current-speed property:

```
DT_PROP(DT_ALIAS(my_serial), current_speed) // 115200
```

Parameters

- `alias` – lowercase-and-underscores alias name.

Returns node identifier for the node with that alias

`DT_INST(inst, compat)`

Get a node identifier for an instance of a compatible.

All nodes with a particular compatible property value are assigned instance numbers, which are zero-based indexes specific to that compatible. You can get a node identifier for these nodes by passing *DT_INST()* an instance number, "inst", along with the lowercase-and-underscores version of the compatible, "compat".

Instance numbers have the following properties:

- for each compatible, instance numbers start at 0 and are contiguous
- exactly one instance number is assigned for each node with a compatible, **including disabled nodes**
- enabled nodes (status property is "okay" or missing) are assigned the instance numbers starting from 0, and disabled nodes have instance numbers which are greater than those of any enabled node

No other guarantees are made. In particular:

- instance numbers **in no way reflect** any numbering scheme that might exist in SoC documentation, node labels or unit addresses, or properties of the /aliases node (use *DT_NODELABEL()* or *DT_ALIAS()* for those)
- there **is no general guarantee** that the same node will have the same instance number between builds, even if you are building the same application again in the same build directory

Example devicetree fragment:

```
serial1: serial@40001000 {
        compatible = "vnd,soc-serial";
        status = "disabled";
        current-speed = <9600>;
        ...
};

serial2: serial@40002000 {
        compatible = "vnd,soc-serial";
        status = "okay";
        current-speed = <57600>;
        ...
};

serial3: serial@40003000 {
        compatible = "vnd,soc-serial";
        current-speed = <115200>;
        ...
};
```

Assuming no other nodes in the devicetree have compatible "vnd,soc-serial", that compatible has nodes with instance numbers 0, 1, and 2.

The nodes serial@40002000 and serial@40003000 are both enabled, so their instance numbers are 0 and 1, but no guarantees are made regarding which node has which instance number.

Since serial@40001000 is the only disabled node, it has instance number 2, since disabled nodes are assigned the largest instance numbers. Therefore:

```
// Could be 57600 or 115200. There is no way to be sure:
// either serial@40002000 or serial@40003000 could
// have instance number 0, so this could be the current-speed
// property of either of those nodes.
DT_PROP(DT_INST(0, vnd_soc_serial), current_speed)
```

(continues on next page)

(continued from previous page)

```
// Could be 57600 or 115200, for the same reason.
// If the above expression expands to 57600, then
// this expands to 115200, and vice-versa.
DT_PROP(DT_INST(1, vnd_soc_serial), current_speed)

// 9600, because there is only one disabled node, and
// disabled nodes are "at the the end" of the instance
// number "list".
DT_PROP(DT_INST(2, vnd_soc_serial), current_speed)
```

Notice how “vnd,soc-serial” in the devicetree becomes vnd_soc_serial (without quotes) in the *DT_INST()* arguments. (As usual, current-speed in the devicetree becomes current_speed as well.)

Nodes whose “compatible” property has multiple values are assigned independent instance numbers for each compatible.

Parameters

- `inst` – instance number for compatible “compat”
- `compat` – lowercase-and-underscores compatible, without quotes

Returns node identifier for the node with that instance number and compatible

`DT_PARENT(node_id)`

Get a node identifier for a parent node.

Example devicetree fragment:

```
parent: parent-node {
        child: child-node {
                ...
        };
};
```

The following are equivalent ways to get the same node identifier:

```
DT_NODELABEL(parent)
DT_PARENT(DT_NODELABEL(child))
```

Parameters

- `node_id` – node identifier

Returns a node identifier for the node’s parent

`DT_GPARENT(node_id)`

Get a node identifier for a grandparent node.

Example devicetree fragment:

```
gparent: grandparent-node {
        parent: parent-node {
                child: child-node { ... }
        };
};
```

The following are equivalent ways to get the same node identifier:

```
DT_GPARENT(DT_NODELABEL(child))
DT_PARENT(DT_PARENT(DT_NODELABEL(child))
```

Parameters

- `node_id` – node identifier

Returns a node identifier for the node's parent's parent

`DT_CHILD(node_id, child)`

Get a node identifier for a child node.

Example devicetree fragment:

```
/ {
        soc-label: soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                        ...
                };
        };
};
```

Example usage with *DT_PROP()* to get the status of the serial@40001000 node:

```
#define SOC_NODE DT_NODELABEL(soc_label)
DT_PROP(DT_CHILD(SOC_NODE, serial_40001000), status) // "okay"
```

Node labels like "serial1" cannot be used as the "child" argument to this macro. Use *DT_NODELABEL()* for that instead.

You can also use *DT_FOREACH_CHILD()* to iterate over node identifiers for all of a node's children.

Parameters

- `node_id` – node identifier
- `child` – lowercase-and-underscores child node name

Returns node identifier for the node with the name referred to by 'child'

`DT_COMPAT_GET_ANY_STATUS_OKAY(compat)`

Get a node identifier for a status "okay" node with a compatible.

Use this if you want to get an arbitrary enabled node with a given compatible, and you do not care which one you get. If any enabled nodes with the given compatible exist, a node identifier for one of them is returned. Otherwise, `DT_INVALID_NODE` is returned.

Example devicetree fragment:

```
node-a {
   compatible = "vnd,device";
   status = "okay";
};

node-b {
   compatible = "vnd,device";
   status = "okay";
};

node-c {
   compatible = "vnd,device";
   status = "disabled";
};
```

Example usage:

```
DT_COMPAT_GET_ANY_STATUS_OKAY(vnd_device)
```

This expands to a node identifier for either `node-a` or `node-b`. It will not expand to a node identifier for `node-c`, because that node does not have status "okay".

Parameters

- `compat` – lowercase-and-underscores compatible, without quotes

Returns node identifier for a node with that compatible, or DT_INVALID_NODE

`DT_NODE_PATH(node_id)`

Get a devicetree node's full path as a string literal.

This returns the path to a node from a node identifier. To get a node identifier from path components instead, use *DT_PATH()*.

Example devicetree fragment:

```
/ {
        soc {
                node: my-node@12345678 { ... };
        };
};
```

Example usage:

DT_NODE_PATH(DT_NODELABEL(node)) // "/soc/my-node@12345678" *DT_NODE_PATH(DT_PATH(soc))* // "/soc" *DT_NODE_PATH(DT_ROOT)* // "/"

Parameters

- `node_id` – node identifier

Returns the node's full path in the devicetree

`DT_NODE_FULL_NAME(node_id)`

Get a devicetree node's name with unit-address as a string literal.

This returns the node name and unit-address from a node identifier.

Example devicetree fragment:

```
/ {
        soc {
                node: my-node@12345678 { ... };
        };
};
```

Example usage:

DT_NODE_FULL_NAME(DT_NODELABEL(node)) // "my-node@12345678"

Parameters

- `node_id` – node identifier

Returns the node's name with unit-address as a string in the devicetree

`DT_SAME_NODE(node_id1, node_id2)`

Do node_id1 and node_id2 refer to the same node?

Both "node_id1" and "node_id2" must be node identifiers for nodes that exist in the devicetree (if unsure, you can check with *DT_NODE_EXISTS()*).

The expansion evaluates to 0 or 1, but may not be a literal integer 0 or 1.

Parameters

- `node_id1` – first node identifer
- `node_id2` – second node identifier

Returns an expression that evaluates to 1 if the node identifiers refer to the same node, and evaluates to 0 otherwise

Property access The following general-purpose macros can be used to access node properties. There are special-purpose APIs for accessing the *reg property* and *interrupts property*.

Property values can be read using these macros even if the node is disabled, as long as it has a matching binding.

group `devicetree-generic-prop`

Defines

`DT_PROP(node_id, prop)`

Get a devicetree property value.

For properties whose bindings have the following types, this macro expands to:

- string: a string literal
- boolean: 0 if the property is false, or 1 if it is true
- int: the property's value as an integer literal
- array, uint8-array, string-array: an initializer expression in braces, whose elements are integer or string literals (like {0, 1, 2}, {"hello", "world"}, etc.)
- phandle: a node identifier for the node with that phandle

A property's type is usually defined by its binding. In some special cases, it has an assumed type defined by the devicetree specification even when no binding is available: "compatible" has type string-array, "status" and "label" have type string, and "interrupt-controller" has type boolean.

For other properties or properties with unknown type due to a missing binding, behavior is undefined.

For usage examples, see *DT_PATH()*, *DT_ALIAS()*, *DT_NODELABEL()*, and *DT_INST()* above.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name

Returns a representation of the property's value

`DT_PROP_LEN(node_id, prop)`

Get a property's logical length.

Here, "length" is a number of elements, which may differ from the property's size in bytes.

The return value depends on the property's type:

- for types array, string-array, and uint8-array, this expands to the number of elements in the array

- for type phandles, this expands to the number of phandles
- for type phandle-array, this expands to the number of phandle and specifier blocks in the property

These properties are handled as special cases:

- reg property: use *DT_NUM_REGS(node_id)* instead
- interrupts property: use *DT_NUM_IRQS(node_id)* instead

It is an error to use this macro with the reg or interrupts properties.

For other properties, behavior is undefined.

Parameters

- `node_id` – node identifier
- `prop` – a lowercase-and-underscores property with a logical length

Returns the property's length

`DT_PROP_LEN_OR(node_id, prop, default_value)`

Like *DT_PROP_LEN()*, but with a fallback to default_value.

If the property is defined (as determined by *DT_NODE_HAS_PROP()*), this expands to *DT_PROP_LEN(node_id, prop)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- `node_id` – node identifier
- `prop` – a lowercase-and-underscores property with a logical length
- `default_value` – a fallback value to expand to

Returns the property's length or the given default value

`DT_PROP_HAS_IDX(node_id, prop, idx)`

Is index "idx" valid for an array type property?

If this returns 1, then *DT_PROP_BY_IDX(node_id, prop, idx)* or *DT_PHA_BY_IDX(node_id, prop, idx, ...)* are valid at index "idx". If it returns 0, it is an error to use those macros with that index.

These properties are handled as special cases:

- reg property: use *DT_REG_HAS_IDX(node_id, idx)* instead
- interrupts property: use *DT_IRQ_HAS_IDX(node_id, idx)* instead

It is an error to use this macro with the reg or interrupts properties.

Parameters

- `node_id` – node identifier
- `prop` – a lowercase-and-underscores property with a logical length
- `idx` – index to check

Returns An expression which evaluates to 1 if "idx" is a valid index into the given property, and 0 otherwise.

DT_PROP_BY_IDX(node_id, prop, idx)

Get the value at index "idx" in an array type property.

It might help to read the argument order as being similar to "node->property[index]".

When the property's binding has type array, string-array, uint8-array, or phandles, this expands to the idx-th array element as an integer, string literal, or node identifier respectively.

These properties are handled as special cases:

- reg property: use *DT_REG_ADDR_BY_IDX()* or *DT_REG_SIZE_BY_IDX()* instead
- interrupts property: use *DT_IRQ_BY_IDX()* instead

For non-array properties, behavior is undefined.

Parameters

- node_id – node identifier
- prop – lowercase-and-underscores property name
- idx – the index to get

Returns a representation of the idx-th element of the property

DT_PROP_OR(node_id, prop, default_value)

Like *DT_PROP()*, but with a fallback to default_value.

If the value exists, this expands to *DT_PROP(node_id, prop)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- node_id – node identifier
- prop – lowercase-and-underscores property name
- default_value – a fallback value to expand to

Returns the property's value or default_value

DT_LABEL(node_id)

Equivalent to *DT_PROP(node_id, label)*

This is a convenience for the Zephyr device API, which uses label properties as *device_get_binding()* arguments.

Parameters

- node_id – node identifier

Returns node's label property value

DT_ENUM_IDX(node_id, prop)

Get a property value's index into its enumeration values.

The return values start at zero.

Example devicetree fragment:

```
usb1: usb@12340000 {
        maximum-speed = "full-speed";
};
usb2: usb@12341000 {
        maximum-speed = "super-speed";
};
```

Example bindings fragment:

```
properties:
  maximum-speed:
    type: string
    enum:
       - "low-speed"
       - "full-speed"
       - "high-speed"
       - "super-speed"
```

Example usage:

```
DT_ENUM_IDX(DT_NODELABEL(usb1), maximum_speed) // 1
DT_ENUM_IDX(DT_NODELABEL(usb2), maximum_speed) // 3
```

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name

Returns zero-based index of the property's value in its enum: list

`DT_ENUM_IDX_OR(node_id, prop, default_idx_value)`

Like *DT_ENUM_IDX()*, but with a fallback to a default enum index.

If the value exists, this expands to its zero based index value thanks to *DT_ENUM_IDX(node_id, prop)*.

Otherwise, this expands to provided default index enum value.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name
- `default_idx_value` – a fallback index value to expand to

Returns zero-based index of the property's value in its enum if present, default_idx_value ohterwise

`DT_STRING_TOKEN(node_id, prop)`

Get a string property's value as a token.

This removes "the quotes" from string-valued properties, and converts non-alphanumeric characters to underscores. That can be useful, for example, when programmatically using the value to form a C variable or code.

DT_STRING_TOKEN() can only be used for properties with string type.

It is an error to use *DT_STRING_TOKEN()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "foo";
};
n2: node-2 {
        prop = "FOO";
}
n3: node-3 {
        prop = "123 foo";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string
```

Example usage:

```
DT_STRING_TOKEN(DT_NODELABEL(n1), prop) // foo
DT_STRING_TOKEN(DT_NODELABEL(n2), prop) // FOO
DT_STRING_TOKEN(DT_NODELABEL(n3), prop) // 123_foo
```

Notice how:

- Unlike C identifiers, the property values may begin with a number. It's the user's responsibility not to use such values as the name of a C identifier.
- The uppercased "FOO" in the DTS remains `FOO` as a token. It is *not* converted to `foo`.
- The whitespace in the DTS "123 foo" string is converted to `123_foo` as a token.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property string name

Returns the value of `prop` as a token, i.e. without any quotes and with special characters converted to underscores

`DT_STRING_UPPER_TOKEN(node_id, prop)`

Like *DT_STRING_TOKEN()*, but uppercased.

This removes "the quotes and capitalize" from string-valued properties, and converts non-alphanumeric characters to underscores. That can be useful, for example, when programmatically using the value to form a C variable or code.

DT_STRING_UPPER_TOKEN() can only be used for properties with string type.

It is an error to use *DT_STRING_UPPER_TOKEN()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "foo";
};
n2: node-2 {
        prop = "123 foo";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string
```

Example usage:

```
DT_STRING_UPPER_TOKEN(DT_NODELABEL(n1), prop) // FOO
DT_STRING_UPPER_TOKEN(DT_NODELABEL(n2), prop) // 123_FOO
```

Notice how:

- Unlike C identifiers, the property values may begin with a number. It's the user's responsibility not to use such values as the name of a C identifier.
- The lowercased "foo" in the DTS becomes `FOO` as a token, i.e. it is uppercased.
- The whitespace in the DTS "123 foo" string is converted to `123_FOO` as a token, i.e. it is uppercased and whitespace becomes an underscore.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property string name

Returns the value of `prop` as a token, i.e. without any quotes and with special characters converted to underscores

`DT_ENUM_TOKEN(node_id, prop)`

Get an enumeration property's value as a token.

This allows you to "remove the quotes" from some string-valued properties. That can be useful, for example, when pasting the values onto some other token to form an enum in C using the ## preprocessor operator.

DT_ENUM_TOKEN() can only be used for properties with string type whose binding has an "enum:". The values in the binding's "enum:" list must be unique after converting non-alphanumeric characters to underscores.

It is an error to use *DT_ENUM_TOKEN()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "foo";
};
n2: node-2 {
        prop = "FOO";
}
n3: node-3 {
        prop = "123 foo";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string
    enum:
       - "foo"
       - "FOO"
       - "123 foo"
```

Example usage:

```
DT_ENUM_TOKEN(DT_NODELABEL(n1), prop) // foo
DT_ENUM_TOKEN(DT_NODELABEL(n2), prop) // FOO
DT_ENUM_TOKEN(DT_NODELABEL(n3), prop) // 123_foo
```

Notice how:

- Unlike C identifiers, the property values may begin with a number. It's the user's responsibility not to use such values as the name of a C identifier.
- The uppercased "FOO" in the DTS remains `FOO` as a token. It is not* converted to `foo`.
- The whitespace in the DTS "123 foo" string is converted to `123_foo` as a token.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name with suitable enumeration of values in its binding

Returns the value of `prop` as a token, i.e. without any quotes and with special characters converted to underscores

`DT_ENUM_UPPER_TOKEN(node_id, prop)`

Like *DT_ENUM_TOKEN()*, but uppercased.

This allows you to "remove the quotes and capitalize" some string-valued properties.

DT_ENUM_UPPER_TOKEN() can only be used for properties with string type whose binding has an "enum:". The values in the binding's "enum:" list must be unique after converting non-alphanumeric characters to underscores and capitalizating any letters.

It is an error to use *DT_ENUM_UPPER_TOKEN()* in other circumstances.

Example devicetree fragment:

```
n1: node-1 {
        prop = "foo";
};
n2: node-2 {
        prop = "123 foo";
};
```

Example bindings fragment:

```
properties:
  prop:
    type: string
    enum:
       - "foo"
       - "123 foo"
```

Example usage:

```
DT_ENUM_TOKEN((DT_NODELABEL(n1), prop) // FOO
DT_ENUM_TOKEN((DT_NODELABEL(n2), prop) // 123_FOO
```

Notice how:

- Unlike C identifiers, the property values may begin with a number. It's the user's responsibility not to use such values as the name of a C identifier.
- The lowercased "foo" in the DTS becomes `FOO` as a token, i.e. it is uppercased.
- The whitespace in the DTS "123 foo" string is converted to `123_FOO` as a token, i.e. it is uppercased and whitespace becomes an underscore.

Parameters

- `node_id` – node identifier

- prop – lowercase-and-underscores property name with suitable enumeration of values in its binding

Returns the value of prop as a capitalized token, i.e. upper case, without any quotes, and with special characters converted to underscores

DT_PROP_BY_PHANDLE_IDX(node_id, phs, idx, prop)

Get a property value from a phandle in a property.

This is a shorthand for:

```
DT_PROP(DT_PHANDLE_BY_IDX(node_id, phs, idx), prop)
```

That is, "prop" is a property of the phandle's node, not a property of "node_id".

Example devicetree fragment:

```
n1: node-1 {
        foo = <&n2 &n3>;
};

n2: node-2 {
        bar = <42>;
};

n3: node-3 {
        baz = <43>;
};
```

Example usage:

```
#define N1 DT_NODELABEL(n1)

DT_PROP_BY_PHANDLE_IDX(N1, foo, 0, bar) // 42
DT_PROP_BY_PHANDLE_IDX(N1, foo, 1, baz) // 43
```

Parameters

- node_id – node identifier
- phs – lowercase-and-underscores property with type "phandle", "phandles", or "phandle-array"
- idx – logical index into "phs", which must be zero if "phs" has type "phandle"
- prop – lowercase-and-underscores property of the phandle's node

Returns the property's value

DT_PROP_BY_PHANDLE_IDX_OR(node_id, phs, idx, prop, default_value)

Like *DT_PROP_BY_PHANDLE_IDX()*, but with a fallback to default_value.

If the value exists, this expands to *DT_PROP_BY_PHANDLE_IDX(node_id, phs,idx, prop)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- node_id – node identifier
- phs – lowercase-and-underscores property with type "phandle", "phandles", or "phandle-array"
- idx – logical index into "phs", which must be zero if "phs" has type "phandle"

- prop – lowercase-and-underscores property of the phandle's node
- default_value – a fallback value to expand to

Returns the property's value

DT_PROP_BY_PHANDLE(node_id, ph, prop)

Get a property value from a phandle's node.

This is equivalent to *DT_PROP_BY_PHANDLE_IDX(node_id, ph, 0, prop)*.

Parameters

- node_id – node identifier
- ph – lowercase-and-underscores property of "node_id" with type "phandle"
- prop – lowercase-and-underscores property of the phandle's node

Returns the property's value

DT_PHA_BY_IDX(node_id, pha, idx, cell)

Get a phandle-array specifier cell value at an index.

It might help to read the argument order as being similar to "node->phandle_array[index].cell". That is, the cell value is in the "pha" property of "node_id", inside the specifier at index "idx".

Example devicetree fragment:

```
gpio0: gpio@... {
        #gpio-cells = <2>;
};

gpio1: gpio@... {
        #gpio-cells = <2>;
};

led: led_0 {
        gpios = <&gpio0 17 0x1>, <&gpio1 5 0x3>;
};
```

Bindings fragment for the gpio0 and gpio1 nodes:

```
gpio-cells:
  - pin
  - flags
```

Above, "gpios" has two elements:

- index 0 has specifier <17 0x1>, so its "pin" cell is 17, and its "flags" cell is 0x1
- index 1 has specifier <5 0x3>, so "pin" is 5 and "flags" is 0x3

Example usage:

```
#define LED DT_NODELABEL(led)

DT_PHA_BY_IDX(LED, gpios, 0, pin)   // 17
DT_PHA_BY_IDX(LED, gpios, 1, flags) // 0x3
```

Parameters

- node_id – node identifier

- `pha` – lowercase-and-underscores property with type "phandle-array"
- `idx` – logical index into "pha"
- `cell` – lowercase-and-underscores cell name within the specifier at "pha" index "idx"

Returns the cell's value

`DT_PHA_BY_IDX_OR(node_id, pha, idx, cell, default_value)`

Like *DT_PHA_BY_IDX()*, but with a fallback to default_value.

If the value exists, this expands to *DT_PHA_BY_IDX(node_id, pha,idx, cell)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- `node_id` – node identifier
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `idx` – logical index into "pha"
- `cell` – lowercase-and-underscores cell name within the specifier at "pha" index "idx"
- `default_value` – a fallback value to expand to

Returns the cell's value or "default_value"

`DT_PHA(node_id, pha, cell)`

Equivalent to *DT_PHA_BY_IDX(node_id, pha, 0, cell)*

Parameters

- `node_id` – node identifier
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `cell` – lowercase-and-underscores cell name

Returns the cell's value

`DT_PHA_OR(node_id, pha, cell, default_value)`

Like *DT_PHA()*, but with a fallback to default_value.

If the value exists, this expands to *DT_PHA(node_id, pha, cell)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- `node_id` – node identifier
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `cell` – lowercase-and-underscores cell name
- `default_value` – a fallback value to expand to

Returns the cell's value or default_value

`DT_PHA_BY_NAME(node_id, pha, name, cell)`

Get a value within a phandle-array specifier by name.

This is like *DT_PHA_BY_IDX()*, except it treats "pha" as a structure where each array element has a name.

It might help to read the argument order as being similar to "node->phandle_struct.name.cell". That is, the cell value is in the "pha" property of "node_id", treated as a data structure where each array element has a name.

Example devicetree fragment:

```
n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Bindings fragment for the "adc1" and "adc2" nodes:

```
io-channel-cells:
  - input
```

Example usage:

```
DT_PHA_BY_NAME(DT_NODELABEL(n), io_channels, sensor, input)  // 10
DT_PHA_BY_NAME(DT_NODELABEL(n), io_channels, bandgap, input) // 20
```

Parameters

- `node_id` – node identifier
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `name` – lowercase-and-underscores name of a specifier in "pha"
- `cell` – lowercase-and-underscores cell name in the named specifier

Returns the cell's value

`DT_PHA_BY_NAME_OR(node_id, pha, name, cell, default_value)`

Like *DT_PHA_BY_NAME()*, but with a fallback to default_value.

If the value exists, this expands to *DT_PHA_BY_NAME(node_id, pha,name, cell)*. The default_value parameter is not expanded in this case.

Otherwise, this expands to default_value.

Parameters

- `node_id` – node identifier
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `name` – lowercase-and-underscores name of a specifier in "pha"
- `cell` – lowercase-and-underscores cell name in the named specifier
- `default_value` – a fallback value to expand to

Returns the cell's value or default_value

`DT_PHANDLE_BY_NAME(node_id, pha, name)`

Get a phandle's node identifier from a phandle array by name.

It might help to read the argument order as being similar to "node->phandle_struct.name.phandle". That is, the phandle array is treated as a structure with named elements. The return value is the node identifier for a phandle inside the structure.

Example devicetree fragment:

```
adc1: adc@... {
        label = "ADC_1";
};

adc2: adc@... {
        label = "ADC_2";
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Above, “io-channels” has two elements:

- the element named “SENSOR” has phandle &adc1
- the element named “BANDGAP” has phandle &adc2

Example usage:

```
#define NODE DT_NODELABEL(n)

DT_LABEL(DT_PHANDLE_BY_NAME(NODE, io_channels, sensor))  // "ADC_1"
DT_LABEL(DT_PHANDLE_BY_NAME(NODE, io_channels, bandgap)) // "ADC_2"
```

Notice how devicetree properties and names are lowercased, and non-alphanumeric characters are converted to underscores.

Parameters

- `node_id` – node identifier
- `pha` – lowercase-and-underscores property with type “phandle-array”
- `name` – lowercase-and-underscores name of an element in “pha”

Returns a node identifier for the node with that phandle

`DT_PHANDLE_BY_IDX(node_id, prop, idx)`

Get a node identifier for a phandle in a property.

When a node’s value at a logical index contains a phandle, this macro returns a node identifier for the node with that phandle.

Therefore, if “prop” has type “phandle”, “idx” must be zero. (A “phandle” type is treated as a “phandles” with a fixed length of 1).

Example devicetree fragment:

```
n1: node-1 {
        foo = <&n2 &n3>;
};

n2: node-2 { ... };
n3: node-3 { ... };
```

Above, “foo” has type phandles and has two elements:

- index 0 has phandle &n2, which is node-2’s phandle
- index 1 has phandle &n3, which is node-3’s phandle

Example usage:

```
#define N1 DT_NODELABEL(n1)

DT_PHANDLE_BY_IDX(N1, foo, 0) // node identifier for node-2
DT_PHANDLE_BY_IDX(N1, foo, 1) // node identifier for node-3
```

Behavior is analogous for phandle-arrays.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name in "node_id" with type "phandle", "phandles" or "phandle-array"
- `idx` – index into "prop"

Returns node identifier for the node with the phandle at that index

`DT_PHANDLE(node_id, prop)`

Get a node identifier for a phandle property's value.

This is equivalent to *DT_PHANDLE_BY_IDX(node_id, prop, 0)*. Its primary benefit is readability when "prop" has type "phandle".

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property of "node_id" with type "phandle"

Returns a node identifier for the node pointed to by "ph"

`reg` **property** Use these APIs instead of *Property access* to access the `reg` property. Because this property's semantics are defined by the devicetree specification, these macros can be used even for nodes without matching bindings.

group `devicetree-reg-prop`

Defines

`DT_NUM_REGS(node_id)`

Get the number of register blocks in the reg property.

Use this instead of *DT_PROP_LEN(node_id, reg)*.

Parameters

- `node_id` – node identifier

Returns Number of register blocks in the node's "reg" property.

`DT_REG_HAS_IDX(node_id, idx)`

Is "idx" a valid register block index?

If this returns 1, then *DT_REG_ADDR_BY_IDX(node_id, idx)* or *DT_REG_SIZE_BY_IDX(node_id, idx)* are valid. If it returns 0, it is an error to use those macros with index "idx".

Parameters

- `node_id` – node identifier
- `idx` – index to check

Returns 1 if "idx" is a valid register block index, 0 otherwise.

DT_REG_ADDR_BY_IDX(node_id, idx)

Get the base address of the register block at index "idx".

Parameters

- node_id – node identifier
- idx – index of the register whose address to return

Returns address of the idx-th register block

DT_REG_SIZE_BY_IDX(node_id, idx)

Get the size of the register block at index "idx".

This is the size of an individual register block, not the total number of register blocks in the property; use *DT_NUM_REGS()* for that.

Parameters

- node_id – node identifier
- idx – index of the register whose size to return

Returns size of the idx-th register block

DT_REG_ADDR(node_id)

Get a node's (only) register block address.

Equivalent to *DT_REG_ADDR_BY_IDX(node_id, 0)*.

Parameters

- node_id – node identifier

Returns node's register block address

DT_REG_SIZE(node_id)

Get a node's (only) register block size.

Equivalent to *DT_REG_SIZE_BY_IDX(node_id, 0)*.

Parameters

- node_id – node identifier

Returns node's only register block's size

DT_REG_ADDR_BY_NAME(node_id, name)

Get a register block's base address by name.

Parameters

- node_id – node identifier
- name – lowercase-and-underscores register specifier name

Returns address of the register block specified by name

DT_REG_SIZE_BY_NAME(node_id, name)

Get a register block's size by name.

Parameters

- node_id – node identifier
- name – lowercase-and-underscores register specifier name

Returns size of the register block specified by name

`interrupts` **property** Use these APIs instead of *Property access* to access the `interrupts` property.

Because this property's semantics are defined by the devicetree specification, some of these macros can be used even for nodes without matching bindings. This does not apply to macros which take cell names as arguments.

group `devicetree-interrupts-prop`

Defines

`DT_NUM_IRQS(node_id)`

Get the number of interrupt sources for the node.

Use this instead of *DT_PROP_LEN(node_id, interrupts)*.

Parameters

- `node_id` – node identifier

Returns Number of interrupt specifiers in the node's "interrupts" property.

`DT_IRQ_HAS_IDX(node_id, idx)`

Is "idx" a valid interrupt index?

If this returns 1, then *DT_IRQ_BY_IDX(node_id, idx)* is valid. If it returns 0, it is an error to use that macro with this index.

Parameters

- `node_id` – node identifier
- `idx` – index to check

Returns 1 if the idx is valid for the interrupt property 0 otherwise.

`DT_IRQ_HAS_CELL_AT_IDX(node_id, idx, cell)`

Does an interrupts property have a named cell specifier at an index? If this returns 1, then *DT_IRQ_BY_IDX(node_id, idx, cell)* is valid. If it returns 0, it is an error to use that macro.

Parameters

- `node_id` – node identifier
- `idx` – index to check
- `cell` – named cell value whose existence to check

Returns 1 if the named cell exists in the interrupt specifier at index idx 0 otherwise.

`DT_IRQ_HAS_CELL(node_id, cell)`

Equivalent to *DT_IRQ_HAS_CELL_AT_IDX(node_id, 0, cell)*

Parameters

- `node_id` – node identifier
- `cell` – named cell value whose existence to check

Returns 1 if the named cell exists in the interrupt specifier at index 0 0 otherwise.

`DT_IRQ_HAS_NAME(node_id, name)`

Does an interrupts property have a named specifier value at an index? If this returns 1, then *DT_IRQ_BY_NAME(node_id, name, cell)* is valid. If it returns 0, it is an error to use that macro.

Parameters

- `node_id` – node identifier

- `name` – lowercase-and-underscores interrupt specifier name

Returns 1 if "name" is a valid named specifier 0 otherwise.

`DT_IRQ_BY_IDX(node_id, idx, cell)`

Get a value within an interrupt specifier at an index.

It might help to read the argument order as being similar to "node->interrupts[index].cell".

This can be used to get information about an individual interrupt when a device generates more than one.

Example devicetree fragment:

```
my-serial: serial@... {
        interrupts = < 33 0 >, < 34 1 >;
};
```

Assuming the node's interrupt domain has "#interrupt-cells = <2>;" and the individual cells in each interrupt specifier are named "irq" and "priority" by the node's binding, here are some examples:

```
#define SERIAL DT_NODELABEL(my_serial)

Example usage                       Value
-------------                       -----
DT_IRQ_BY_IDX(SERIAL, 0, irq)          33
DT_IRQ_BY_IDX(SERIAL, 0, priority)      0
DT_IRQ_BY_IDX(SERIAL, 1, irq,          34
DT_IRQ_BY_IDX(SERIAL, 1, priority)      1
```

Parameters

- `node_id` – node identifier
- `idx` – logical index into the interrupt specifier array
- `cell` – cell name specifier

Returns the named value at the specifier given by the index

`DT_IRQ_BY_NAME(node_id, name, cell)`

Get a value within an interrupt specifier by name.

It might help to read the argument order as being similar to "node->interrupts.name.cell".

This can be used to get information about an individual interrupt when a device generates more than one, if the bindings give each interrupt specifier a name.

Parameters

- `node_id` – node identifier
- `name` – lowercase-and-underscores interrupt specifier name
- `cell` – cell name specifier

Returns the named value at the specifier given by the index

`DT_IRQ(node_id, cell)`

Get an interrupt specifier's value Equivalent to *DT_IRQ_BY_IDX(node_id, 0, cell)*.

Parameters

- `node_id` – node identifier
- `cell` – cell name specifier

Returns the named value at that index

DT_IRQN(node_id)

Get a node's (only) irq number.

Equivalent to *DT_IRQ(node_id, irq)*. This is provided as a convenience for the common case where a node generates exactly one interrupt, and the IRQ number is in a cell named "irq".

Parameters

- node_id – node identifier

Returns the interrupt number for the node's only interrupt

For-each macros There is currently only one "generic" for-each macro, *DT_FOREACH_CHILD()*, which allows iterating over the children of a devicetree node.

There are special-purpose for-each macros, like *DT_INST_FOREACH_STATUS_OKAY()*, but these require DT_DRV_COMPAT to be defined before use.

group devicetree-generic-foreach

Defines

DT_FOREACH_CHILD(node_id, fn)

Invokes "fn" for each child of "node_id".

The macro "fn" must take one parameter, which will be the node identifier of a child node of "node_id".

Example devicetree fragment:

```
n: node {
        child-1 {
                label = "foo";
        };
        child-2 {
                label = "bar";
        };
};
```

Example usage:

```
#define LABEL_AND_COMMA(node_id) DT_LABEL(node_id),

const char *child_labels[] = {
    DT_FOREACH_CHILD(DT_NODELABEL(n), LABEL_AND_COMMA)
};
```

This expands to:

```
const char *child_labels[] = {
    "foo", "bar",
};
```

Parameters

- node_id – node identifier
- fn – macro to invoke

DT_FOREACH_CHILD_VARGS(node_id, fn, ...)

Invokes "fn" for each child of "node_id" with multiple arguments.

The macro "fn" takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

See also:

DT_FOREACH_CHILD

Parameters

- node_id – node identifier
- fn – macro to invoke
- ... – variable number of arguments to pass to fn

DT_FOREACH_CHILD_STATUS_OKAY(node_id, fn)

Call "fn" on the child nodes with status "okay".

The macro "fn" should take one argument, which is the node identifier for the child node.

As usual, both a missing status and an "ok" status are treated as "okay".

Parameters

- node_id – node identifier
- fn – macro to invoke

DT_FOREACH_CHILD_STATUS_OKAY_VARGS(node_id, fn, ...)

Call "fn" on the child nodes with status "okay" with multiple arguments.

The macro "fn" takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

As usual, both a missing status and an "ok" status are treated as "okay".

See also:

DT_FOREACH_CHILD_STATUS_OKAY

Parameters

- node_id – node identifier
- fn – macro to invoke
- ... – variable number of arguments to pass to fn

DT_FOREACH_PROP_ELEM(node_id, prop, fn)

Invokes "fn" for each element in the value of property "prop".

The macro "fn" must take three parameters: fn(node_id, prop, idx). "node_id" and "prop" are the same as what is passed to DT_FOREACH_PROP_ELEM, and "idx" is the current index into the array. The "idx" values are integer literals starting from 0.

Example devicetree fragment:

```
n: node {
        my-ints = <1 2 3>;
};
```

Example usage:

```
#define TIMES_TWO(node_id, prop, idx) \
        (2 * DT_PROP_BY_IDX(node_id, prop, idx)),

int array[] = {
        DT_FOREACH_PROP_ELEM(DT_NODELABEL(n), my_ints, TIMES_TWO)
};
```

This expands to:

```
int array[] = {
        (2 * 1), (2 * 2), (2 * 3),
};
```

In general, this macro expands to:

```
fn(node_id, prop, 0) fn(node_id, prop, 1) [...] fn(node_id, prop, n-1)
```

where "n" is the number of elements in "prop", as it would be returned by *DT_PROP_LEN(node_id, prop)*.

The "prop" argument must refer to a property with type string, array, uint8-array, string-array, phandles, or phandle-array. It is an error to use this macro with properties of other types.

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name
- `fn` – macro to invoke

`DT_FOREACH_PROP_ELEM_VARGS(node_id, prop, fn, ...)`

Invokes "fn" for each element in the value of property "prop" with multiple arguments.

The macro "fn" must take multiple parameters: fn(node_id, prop, idx, ...). "node_id" and "prop" are the same as what is passed to DT_FOREACH_PROP_ELEM, and "idx" is the current index into the array. The "idx" values are integer literals starting from 0. The remaining arguments are passed-in by the caller.

See also:

DT_FOREACH_PROP_ELEM

Parameters

- `node_id` – node identifier
- `prop` – lowercase-and-underscores property name
- `fn` – macro to invoke
- `...` – variable number of arguments to pass to fn

`DT_FOREACH_STATUS_OKAY(compat, fn)`

Call "fn" on all nodes with compatible DT_DRV_COMPAT and status "okay".

This macro expands to:

```
fn(node_id_1) fn(node_id_2) ... fn(node_id_n)
```

where each "node_id_<i>" is a node identifier for some node with compatible "compat" and status "okay". Whitespace is added between expansions as shown above.

Example devicetree fragment:

```
/ {
        a {
                compatible = "foo";
                status = "okay";
        };
        b {
                compatible = "foo";
                status = "disabled";
        };
        c {
                compatible = "foo";
        };
};
```

Example usage:

```
DT_FOREACH_STATUS_OKAY(foo, DT_NODE_PATH)
```

This expands to one of the following:

```
"/a" "/c"
"/c" "/a"
```

“One of the following” is because no guarantees are made about the order that node identifiers are passed to “fn” in the expansion.

(The “/c” string literal is present because a missing status property is always treated as if the status were set to “okay”.)

Note also that “fn” is responsible for adding commas, semicolons, or other terminators as needed.

Parameters

- `compat` – lowercase-and-underscores devicetree compatible
- `fn` – Macro to call for each enabled node. Must accept a node_id as its only parameter.

`DT_FOREACH_STATUS_OKAY_VARGS(compat, fn, ...)`

Invokes “fn” for each status “okay” node of a compatible with multiple arguments.

This is like *DT_FOREACH_STATUS_OKAY()* except you can also pass additional arguments to “fn”.

Example devicetree fragment:

```
/ {
        a {
                compatible = "foo";
                val = <3>;
        };
        b {
                compatible = "foo";
                val = <4>;
        };
};
```

Example usage:

```
#define MY_FN(node_id, operator) DT_PROP(node_id, val) operator
x = DT_FOREACH_STATUS_OKAY_VARGS(foo, MY_FN, +) 0;
```

This expands to one of the following:

```
x = 3 + 4 + 0;
x = 4 + 3 + 0;
```

i.e. it sets x to 7. As with *DT_FOREACH_STATUS_OKAY()*, there are no guarantees about the order nodes appear in the expansion.

Parameters

- `compat` – lowercase-and-underscores devicetree compatible
- `fn` – Macro to call for each enabled node. Must accept a node_id as its only parameter.
- `...` – Additional arguments to pass to "fn"

Existence checks This section documents miscellaneous macros that can be used to test if a node exists, how many nodes of a certain type exist, whether a node has certain properties, etc. Some macros used for special purposes (such as *DT_IRQ_HAS_IDX()* and all macros which require `DT_DRV_COMPAT`) are documented elsewhere on this page.

group `devicetree-generic-exist`

Defines

`DT_NODE_EXISTS(node_id)`

Does a node identifier refer to a node?

Tests whether a node identifier refers to a node which exists, i.e. is defined in the devicetree.

It doesn't matter whether or not the node has a matching binding, or what the node's status value is. This is purely a check of whether the node exists at all.

Parameters

- `node_id` – a node identifier

Returns 1 if the node identifier refers to a node, 0 otherwise.

`DT_NODE_HAS_STATUS(node_id, status)`

Does a node identifier refer to a node with a status?

Example uses:

```
DT_NODE_HAS_STATUS(DT_PATH(soc, i2c_12340000), okay)
DT_NODE_HAS_STATUS(DT_PATH(soc, i2c_12340000), disabled)
```

Tests whether a node identifier refers to a node which:

- exists in the devicetree, and
- has a status property matching the second argument (except that either a missing status or an "ok" status in the devicetree is treated as if it were "okay" instead)

Parameters

- `node_id` – a node identifier
- `status` – a status as one of the tokens okay or disabled, not a string

Returns 1 if the node has the given status, 0 otherwise.

DT_HAS_COMPAT_STATUS_OKAY(compat)

Does the devicetree have a status "okay" node with a compatible?

Test for whether the devicetree has any nodes with status "okay" and the given compatible. That is, this returns 1 if and only if there is at least one "node_id" for which both of these expressions return 1:

```
DT_NODE_HAS_STATUS(node_id, okay)
DT_NODE_HAS_COMPAT(node_id, compat)
```

As usual, both a missing status and an "ok" status are treated as "okay".

Parameters

- compat – lowercase-and-underscores compatible, without quotes

Returns 1 if both of the above conditions are met, 0 otherwise

DT_NUM_INST_STATUS_OKAY(compat)

Get the number of instances of a given compatible with status "okay".

Parameters

- compat – lowercase-and-underscores compatible, without quotes

Returns Number of instances with status "okay"

DT_NODE_HAS_COMPAT(node_id, compat)

Does a devicetree node match a compatible?

Example devicetree fragment:

```
n: node {
        compatible = "vnd,specific-device", "generic-device";
}
```

Example usages which evaluate to 1:

```
DT_NODE_HAS_COMPAT(DT_NODELABEL(n), vnd_specific_device)
DT_NODE_HAS_COMPAT(DT_NODELABEL(n), generic_device)
```

This macro only uses the value of the compatible property. Whether or not a particular compatible has a matching binding has no effect on its value, nor does the node's status.

Parameters

- node_id – node identifier
- compat – lowercase-and-underscores compatible, without quotes

Returns 1 if the node's compatible property contains compat, 0 otherwise.

DT_NODE_HAS_COMPAT_STATUS(node_id, compat, status)

Does a devicetree node have a compatible and status?

This is equivalent to:

```
(DT_NODE_HAS_COMPAT(node_id, compat) &&
 DT_NODE_HAS_STATUS(node_id, status))
```

Parameters

- node_id – node identifier
- compat – lowercase-and-underscores compatible, without quotes
- status – okay or disabled as a token, not a string

DT_NODE_HAS_PROP(node_id, prop)

Does a devicetree node have a property?

Tests whether a devicetree node has a property defined.

This tests whether the property is defined at all, not whether a boolean property is true or false. To get a boolean property's truth value, use *DT_PROP(node_id, prop)* instead.

Parameters

- node_id – node identifier
- prop – lowercase-and-underscores property name

Returns 1 if the node has the property, 0 otherwise.

DT_PHA_HAS_CELL_AT_IDX(node_id, pha, idx, cell)

Does a phandle array have a named cell specifier at an index?

If this returns 1, then the phandle-array property "pha" has a cell named "cell" at index "idx", and therefore *DT_PHA_BY_IDX(node_id,pha, idx, cell)* is valid. If it returns 0, it's an error to use *DT_PHA_BY_IDX()* with the same arguments.

Parameters

- node_id – node identifier
- pha – lowercase-and-underscores property with type "phandle-array"
- idx – index to check within "pha"
- cell – lowercase-and-underscores cell name whose existence to check at index "idx"

Returns 1 if the named cell exists in the specifier at index idx, 0 otherwise.

DT_PHA_HAS_CELL(node_id, pha, cell)

Equivalent to *DT_PHA_HAS_CELL_AT_IDX(node_id, pha, 0, cell)*

Parameters

- node_id – node identifier
- pha – lowercase-and-underscores property with type "phandle-array"
- cell – lowercase-and-underscores cell name whose existence to check at index "idx"

Returns 1 if the named cell exists in the specifier at index 0, 0 otherwise.

Inter-node dependencies The devicetree.h API has some support for tracking dependencies between nodes. Dependency tracking relies on a binary "depends on" relation between devicetree nodes, which is defined as the transitive closure of the following "directly depends on" relation:

- every non-root node directly depends on its parent node
- a node directly depends on any nodes its properties refer to by phandle
- a node directly depends on its interrupt-parent if it has an interrupts property

A *dependency ordering* of a devicetree is a list of its nodes, where each node n appears earlier in the list than any nodes that depend on n. A node's *dependency ordinal* is then its zero-based index in that list. Thus, for two distinct devicetree nodes n1 and n2 with dependency ordinals d1 and d2, we have:

- d1 != d2
- if n1 depends on n2, then d1 > d2
- d1 > d2 does **not** necessarily imply that n1 depends on n2

The Zephyr build system chooses a dependency ordering of the final devicetree and assigns a dependency ordinal to each node. Dependency related information can be accessed using the following macros. The exact dependency ordering chosen is an implementation detail, but cyclic dependencies are detected and cause errors, so it's safe to assume there are none when using these macros.

There are instance number-based conveniences as well; see *DT_INST_DEP_ORD()* and subsequent documentation.

group devicetree-dep-ord

Defines

DT_DEP_ORD(node_id)

Get a node's dependency ordinal.

Parameters

- node_id – Node identifier

Returns the node's dependency ordinal as an integer literal

DT_REQUIRES_DEP_ORDS(node_id)

Get a list of dependency ordinals of a node's direct dependencies.

There is a comma after each ordinal in the expansion, **including** the last one:

```
DT_REQUIRES_DEP_ORDS(my_node) // required_ord_1, ..., required_ord_n,
```

The one case *DT_REQUIRES_DEP_ORDS()* expands to nothing is when given the root node identifier DT_ROOT as argument. The root has no direct dependencies; every other node at least depends on its parent.

Parameters

- node_id – Node identifier

Returns a list of dependency ordinals, with each ordinal followed by a comma (,), or an empty expansion

DT_SUPPORTS_DEP_ORDS(node_id)

Get a list of dependency ordinals of what depends directly on a node.

There is a comma after each ordinal in the expansion, **including** the last one:

```
DT_SUPPORTS_DEP_ORDS(my_node) // supported_ord_1, ..., supported_ord_n,
```

DT_SUPPORTS_DEP_ORDS() may expand to nothing. This happens when node_id refers to a leaf node that nothing else depends on.

Parameters

- node_id – Node identifier

Returns a list of dependency ordinals, with each ordinal followed by a comma (,), or an empty expansion

DT_INST_DEP_ORD(inst)

Get a DT_DRV_COMPAT instance's dependency ordinal.

Equivalent to *DT_DEP_ORD(DT_DRV_INST(inst))*.

Parameters

- inst – instance number

Returns The instance's dependency ordinal

DT_INST_REQUIRES_DEP_ORDS(inst)

Get a list of dependency ordinals of a DT_DRV_COMPAT instance's direct dependencies.

Equivalent to *DT_REQUIRES_DEP_ORDS(DT_DRV_INST(inst))*.

Parameters

- inst – instance number

Returns a list of dependency ordinals for the nodes the instance depends on directly

DT_INST_SUPPORTS_DEP_ORDS(inst)

Get a list of dependency ordinals of what depends directly on a DT_DRV_COMPAT instance.

Equivalent to *DT_SUPPORTS_DEP_ORDS(DT_DRV_INST(inst))*.

Parameters

- inst – instance number

Returns a list of node identifiers for the nodes that depend directly on the instance

Bus helpers Zephyr's devicetree bindings language supports a `bus:` key which allows bindings to declare that nodes with a given compatible describe system buses. In this case, child nodes are considered to be on a bus of the given type, and the following APIs may be used.

group devicetree-generic-bus

Defines

DT_BUS(node_id)

Node's bus controller.

Get the node identifier of the node's bus controller. This can be used with *DT_PROP()* to get properties of the bus controller.

It is an error to use this with nodes which do not have bus controllers.

Example devicetree fragment:

```
i2c@deadbeef {
        label = "I2C_CTLR";
        status = "okay";
        clock-frequency = < 100000 >;

        i2c_device: accelerometer@12 {
                ...
        };
};
```

Example usage:

```
DT_PROP(DT_BUS(DT_NODELABEL(i2c_device)), clock_frequency) // 100000
```

Parameters

- node_id – node identifier

Returns a node identifier for the node's bus controller

DT_BUS_LABEL(node_id)

Node's bus controller's label property.

Parameters

- node_id – node identifier

Returns the label property of the node's bus controller *DT_BUS(node)*

DT_ON_BUS(node_id, bus)

Is a node on a bus of a given type?

Example devicetree overlay:

```
&i2c0 {
        temp: temperature-sensor@76 {
                compatible = "vnd,some-sensor";
                reg = <0x76>;
        };
};
```

Example usage, assuming "i2c0" is an I2C bus controller node, and therefore "temp" is on an I2C bus:

```
DT_ON_BUS(DT_NODELABEL(temp), i2c) // 1
DT_ON_BUS(DT_NODELABEL(temp), spi) // 0
```

Parameters

- node_id – node identifier
- bus – lowercase-and-underscores bus type as a C token (i.e. without quotes)

Returns 1 if the node is on a bus of the given type, 0 otherwise

Instance-based APIs

These are recommended for use within device drivers. To use them, define DT_DRV_COMPAT to the lowercase-and-underscores compatible the device driver implements support for. Here is an example devicetree fragment:

```
serial@40001000 {
        compatible = "vnd,serial";
        status = "okay";
        current-speed = <115200>;
};
```

Example usage, assuming serial@40001000 is the only enabled node with compatible "vnd,serial":

```
#define DT_DRV_COMPAT vnd_serial
DT_DRV_INST(0)                  // node identifier for serial@40001000
DT_INST_PROP(0, current_speed)  // 115200
```

Warning: Be careful making assumptions about instance numbers. See *DT_INST()* for the API guarantees.

As shown above, the DT_INST_* APIs are conveniences for addressing nodes by instance number. They are almost all defined in terms of one of the *Generic APIs*. The equivalent generic API can be found by removing INST_ from the macro name. For example, DT_INST_PROP(inst,

prop) is equivalent to DT_PROP(DT_DRV_INST(inst), prop). Similarly, DT_INST_REG_ADDR(inst) is equivalent to DT_REG_ADDR(DT_DRV_INST(inst)), and so on. There are some exceptions: *DT_ANY_INST_ON_BUS_STATUS_OKAY()* and *DT_INST_FOREACH_STATUS_OKAY()* are special-purpose helpers without straightforward generic equivalents.

Since DT_DRV_INST() requires DT_DRV_COMPAT to be defined, it's an error to use any of these without that macro defined.

Note that there are also helpers available for specific hardware; these are documented in *Hardware specific APIs*.

group devicetree-inst

Defines

DT_DRV_INST(inst)

Node identifier for an instance of a DT_DRV_COMPAT compatible.

Parameters

- inst – instance number

Returns a node identifier for the node with DT_DRV_COMPAT compatible and instance number "inst"

DT_INST_FOREACH_CHILD(inst, fn)

Call "fn" on all child nodes of *DT_DRV_INST(inst)*.

The macro "fn" should take one argument, which is the node identifier for the child node.

See also:

DT_FOREACH_CHILD

Parameters

- inst – instance number
- fn – macro to invoke on each child node identifier

DT_INST_FOREACH_CHILD_VARGS(inst, fn, ...)

Call "fn" on all child nodes of *DT_DRV_INST(inst)*.

The macro "fn" takes multiple arguments. The first should be the node identifier for the child node. The remaining are passed-in by the caller.

See also:

DT_FOREACH_CHILD

Parameters

- inst – instance number
- fn – macro to invoke on each child node identifier
- ... – variable number of arguments to pass to fn

DT_INST_PROP(inst, prop)

Get a DT_DRV_COMPAT instance property.

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name

Returns a representation of the property's value

DT_INST_PROP_LEN(inst, prop)

Get a DT_DRV_COMPAT property length.

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name

Returns logical length of the property

DT_INST_PROP_HAS_IDX(inst, prop, idx)

Is index "idx" valid for an array type property on a DT_DRV_COMPAT instance?

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name
- idx – index to check

Returns 1 if "idx" is a valid index into the given property, 0 otherwise.

DT_INST_PROP_BY_IDX(inst, prop, idx)

Get a DT_DRV_COMPAT element value in an array property.

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name
- idx – the index to get

Returns a representation of the idx-th element of the property

DT_INST_PROP_OR(inst, prop, default_value)

Like *DT_INST_PROP()*, but with a fallback to default_value.

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name
- default_value – a fallback value to expand to

Returns *DT_INST_PROP(inst, prop)* or default_value

DT_INST_LABEL(inst)

Get a DT_DRV_COMPAT instance's "label" property.

Parameters

- inst – instance number

Returns instance's label property value

DT_INST_PROP_BY_PHANDLE(inst, ph, prop)

Get a DT_DRV_COMPAT instance's property value from a phandle's node.

Parameters

- inst – instance number
- ph – lowercase-and-underscores property of "inst" with type "phandle"
- prop – lowercase-and-underscores property of the phandle's node

Returns the value of "prop" as described in the *DT_PROP()* documentation

DT_INST_PROP_BY_PHANDLE_IDX(inst, phs, idx, prop)

Get a DT_DRV_COMPAT instance's property value from a phandle in a property.

Parameters

- inst – instance number
- phs – lowercase-and-underscores property with type "phandle", "phandles", or "phandle-array"
- idx – logical index into "phs", which must be zero if "phs" has type "phandle"
- prop – lowercase-and-underscores property of the phandle's node

Returns the value of "prop" as described in the *DT_PROP()* documentation

DT_INST_PHA_BY_IDX(inst, pha, idx, cell)

Get a DT_DRV_COMPAT instance's phandle-array specifier value at an index.

Parameters

- inst – instance number
- pha – lowercase-and-underscores property with type "phandle-array"
- idx – logical index into the property "pha"
- cell – binding's cell name within the specifier at index "idx"

Returns the value of the cell inside the specifier at index "idx"

DT_INST_PHA_BY_IDX_OR(inst, pha, idx, cell, default_value)

Like *DT_INST_PHA_BY_IDX()*, but with a fallback to default_value.

Parameters

- inst – instance number
- pha – lowercase-and-underscores property with type "phandle-array"
- idx – logical index into the property "pha"
- cell – binding's cell name within the specifier at index "idx"
- default_value – a fallback value to expand to

Returns *DT_INST_PHA_BY_IDX(inst, pha, idx, cell)* or default_value

DT_INST_PHA(inst, pha, cell)

Get a DT_DRV_COMPAT instance's phandle-array specifier value Equivalent to *DT_INST_PHA_BY_IDX(inst, pha, 0, cell)*

Parameters

- inst – instance number
- pha – lowercase-and-underscores property with type "phandle-array"
- cell – binding's cell name for the specifier at "pha" index 0

Returns the cell value

`DT_INST_PHA_OR`(inst, pha, cell, default_value)

Like *DT_INST_PHA()*, but with a fallback to default_value.

Parameters

- `inst` – instance number
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `cell` – binding's cell name for the specifier at "pha" index 0
- `default_value` – a fallback value to expand to

Returns *DT_INST_PHA(inst, pha, cell)* or default_value

`DT_INST_PHA_BY_NAME`(inst, pha, name, cell)

Get a DT_DRV_COMPAT instance's value within a phandle-array specifier by name.

Parameters

- `inst` – instance number
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `name` – lowercase-and-underscores name of a specifier in "pha"
- `cell` – binding's cell name for the named specifier

Returns the cell value

`DT_INST_PHA_BY_NAME_OR`(inst, pha, name, cell, default_value)

Like *DT_INST_PHA_BY_NAME()*, but with a fallback to default_value.

Parameters

- `inst` – instance number
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `name` – lowercase-and-underscores name of a specifier in "pha"
- `cell` – binding's cell name for the named specifier
- `default_value` – a fallback value to expand to

Returns *DT_INST_PHA_BY_NAME(inst, pha, name, cell)* or default_value

`DT_INST_PHANDLE_BY_NAME`(inst, pha, name)

Get a DT_DRV_COMPAT instance's phandle node identifier from a phandle array by name.

Parameters

- `inst` – instance number
- `pha` – lowercase-and-underscores property with type "phandle-array"
- `name` – lowercase-and-underscores name of an element in "pha"

Returns node identifier for the phandle at the element named "name"

`DT_INST_PHANDLE_BY_IDX`(inst, prop, idx)

Get a DT_DRV_COMPAT instance's node identifier for a phandle in a property.

Parameters

- `inst` – instance number
- `prop` – lowercase-and-underscores property name in "inst" with type "phandle", "phandles" or "phandle-array"
- `idx` – index into "prop"

Returns a node identifier for the phandle at index “idx” in “prop”

DT_INST_PHANDLE(inst, prop)

Get a DT_DRV_COMPAT instance’s node identifier for a phandle property’s value.

Parameters

- inst – instance number
- prop – lowercase-and-underscores property of “inst” with type “phandle”

Returns a node identifier for the node pointed to by “ph”

DT_INST_REG_HAS_IDX(inst, idx)

is “idx” a valid register block index on a DT_DRV_COMPAT instance?

Parameters

- inst – instance number
- idx – index to check

Returns 1 if “idx” is a valid register block index, 0 otherwise.

DT_INST_REG_ADDR_BY_IDX(inst, idx)

Get a DT_DRV_COMPAT instance’s idx-th register block’s address.

Parameters

- inst – instance number
- idx – index of the register whose address to return

Returns address of the instance’s idx-th register block

DT_INST_REG_SIZE_BY_IDX(inst, idx)

Get a DT_DRV_COMPAT instance’s idx-th register block’s size.

Parameters

- inst – instance number
- idx – index of the register whose size to return

Returns size of the instance’s idx-th register block

DT_INST_REG_ADDR_BY_NAME(inst, name)

Get a DT_DRV_COMPAT’s register block address by name.

Parameters

- inst – instance number
- name – lowercase-and-underscores register specifier name

Returns address of the register block with the given name

DT_INST_REG_SIZE_BY_NAME(inst, name)

Get a DT_DRV_COMPAT’s register block size by name.

Parameters

- inst – instance number
- name – lowercase-and-underscores register specifier name

Returns size of the register block with the given name

DT_INST_REG_ADDR(inst)

Get a DT_DRV_COMPAT's (only) register block address.

Parameters

- inst – instance number

Returns instance's register block address

DT_INST_REG_SIZE(inst)

Get a DT_DRV_COMPAT's (only) register block size.

Parameters

- inst – instance number

Returns instance's register block size

DT_INST_IRQ_BY_IDX(inst, idx, cell)

Get a DT_DRV_COMPAT interrupt specifier value at an index.

Parameters

- inst – instance number
- idx – logical index into the interrupt specifier array
- cell – cell name specifier

Returns the named value at the specifier given by the index

DT_INST_IRQ_BY_NAME(inst, name, cell)

Get a DT_DRV_COMPAT interrupt specifier value by name.

Parameters

- inst – instance number
- name – lowercase-and-underscores interrupt specifier name
- cell – cell name specifier

Returns the named value at the specifier given by the index

DT_INST_IRQ(inst, cell)

Get a DT_DRV_COMPAT interrupt specifier's value.

Parameters

- inst – instance number
- cell – cell name specifier

Returns the named value at that index

DT_INST_IRQN(inst)

Get a DT_DRV_COMPAT's (only) irq number.

Parameters

- inst – instance number

Returns the interrupt number for the node's only interrupt

DT_INST_BUS(inst)

Get a DT_DRV_COMPAT's bus node identifier.

Parameters

- inst – instance number

Returns node identifier for the instance's bus node

DT_INST_BUS_LABEL(inst)

Get a DT_DRV_COMPAT's bus node's label property.

Parameters

- inst – instance number

Returns the label property of the instance's bus controller

DT_INST_ON_BUS(inst, bus)

Test if a DT_DRV_COMPAT's bus type is a given type.

Parameters

- inst – instance number
- bus – a binding's bus type as a C token, lowercased and without quotes

Returns 1 if the given instance is on a bus of the given type, 0 otherwise

DT_ANY_INST_ON_BUS_STATUS_OKAY(bus)

Test if any DT_DRV_COMPAT node is on a bus of a given type and has status okay.

This is a special-purpose macro which can be useful when writing drivers for devices which can appear on multiple buses. One example is a sensor device which may be wired on an I2C or SPI bus.

Example devicetree overlay:

```
&i2c0 {
       temp: temperature-sensor@76 {
                compatible = "vnd,some-sensor";
                reg = <0x76>;
       };
};
```

Example usage, assuming "i2c0" is an I2C bus controller node, and therefore "temp" is on an I2C bus:

```
#define DT_DRV_COMPAT vnd_some_sensor

DT_ANY_INST_ON_BUS_STATUS_OKAY(i2c) // 1
```

Parameters

- bus – a binding's bus type as a C token, lowercased and without quotes

Returns 1 if any enabled node with that compatible is on that bus type, 0 otherwise

DT_INST_FOREACH_STATUS_OKAY(fn)

Call "fn" on all nodes with compatible DT_DRV_COMPAT and status "okay".

This macro calls "fn(inst)" on each "inst" number that refers to a node with status "okay". Whitespace is added between invocations.

Example devicetree fragment:

```
a {
        compatible = "vnd,device";
        status = "okay";
        label = "DEV_A";
};

b {
```

(continues on next page)

(continued from previous page)

```
        compatible = "vnd,device";
        status = "okay";
        label = "DEV_B";
};

c {
        compatible = "vnd,device";
        status = "disabled";
        label = "DEV_C";
};
```

Example usage:

```
#define DT_DRV_COMPAT vnd_device
#define MY_FN(inst) DT_INST_LABEL(inst),

DT_INST_FOREACH_STATUS_OKAY(MY_FN)
```

This expands to:

```
MY_FN(0) MY_FN(1)
```

and from there, to either this:

```
"DEV_A", "DEV_B",
```

or this:

```
"DEV_B", "DEV_A",
```

No guarantees are made about the order that a and b appear in the expansion.

Note that "fn" is responsible for adding commas, semicolons, or other separators or terminators.

Device drivers should use this macro whenever possible to instantiate a struct device for each enabled node in the devicetree of the driver's compatible DT_DRV_COMPAT.

Parameters

- `fn` – Macro to call for each enabled node. Must accept an instance number as its only parameter.

`DT_INST_FOREACH_STATUS_OKAY_VARGS(fn, ...)`

Call "fn" on all nodes with compatible DT_DRV_COMPAT and status "okay" with multiple arguments.

See also:

DT_INST_FOREACH_STATUS_OKAY

Parameters

- `fn` – Macro to call for each enabled node. Must accept an instance number as its only parameter.
- `...` – variable number of arguments to pass to fn

DT_INST_FOREACH_PROP_ELEM(inst, prop, fn)

Invokes "fn" for each element of property "prop" for a DT_DRV_COMPAT instance.

Equivalent to *DT_FOREACH_PROP_ELEM(DT_DRV_INST(inst), prop, fn)*.

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name
- fn – macro to invoke

DT_INST_FOREACH_PROP_ELEM_VARGS(inst, prop, fn, ...)

Invokes "fn" for each element of property "prop" for a DT_DRV_COMPAT instance with multiple arguments.

Equivalent to DT_FOREACH_PROP_ELEM_VARGS(*DT_DRV_INST(inst)*, prop, fn, **VA_ARGS**)

See also:

DT_INST_FOREACH_PROP_ELEM

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name
- fn – macro to invoke
- ... – variable number of arguments to pass to fn

DT_INST_NODE_HAS_PROP(inst, prop)

Does a DT_DRV_COMPAT instance have a property?

Parameters

- inst – instance number
- prop – lowercase-and-underscores property name

Returns 1 if the instance has the property, 0 otherwise.

DT_INST_PHA_HAS_CELL_AT_IDX(inst, pha, idx, cell)

Does a phandle array have a named cell specifier at an index for a DT_DRV_COMPAT instance?

Parameters

- inst – instance number
- pha – lowercase-and-underscores property with type "phandle-array"
- idx – index to check
- cell – named cell value whose existence to check

Returns 1 if the named cell exists in the specifier at index idx, 0 otherwise.

DT_INST_PHA_HAS_CELL(inst, pha, cell)

Does a phandle array have a named cell specifier at index 0 for a DT_DRV_COMPAT instance?

Parameters

- inst – instance number
- pha – lowercase-and-underscores property with type "phandle-array"
- cell – named cell value whose existence to check

Returns 1 if the named cell exists in the specifier at index 0, 0 otherwise.

`DT_INST_IRQ_HAS_IDX(inst, idx)`

is index valid for interrupt property on a DT_DRV_COMPAT instance?

Parameters

- `inst` – instance number
- `idx` – logical index into the interrupt specifier array

Returns 1 if the idx is valid for the interrupt property 0 otherwise.

`DT_INST_IRQ_HAS_CELL_AT_IDX(inst, idx, cell)`

Does a DT_DRV_COMPAT instance have an interrupt named cell specifier?

Parameters

- `inst` – instance number
- `idx` – index to check
- `cell` – named cell value whose existence to check

Returns 1 if the named cell exists in the interrupt specifier at index idx 0 otherwise.

`DT_INST_IRQ_HAS_CELL(inst, cell)`

Does a DT_DRV_COMPAT instance have an interrupt value?

Parameters

- `inst` – instance number
- `cell` – named cell value whose existence to check

Returns 1 if the named cell exists in the interrupt specifier at index 0 0 otherwise.

`DT_INST_IRQ_HAS_NAME(inst, name)`

Does a DT_DRV_COMPAT instance have an interrupt value?

Parameters

- `inst` – instance number
- `name` – lowercase-and-underscores interrupt specifier name

Returns 1 if “name” is a valid named specifier

Hardware specific APIs

The following APIs can also be used by including `<devicetree.h>`; no additional include is needed.

Clocks These conveniences may be used for nodes which describe clock sources, and properties related to them.

group `devicetree-clocks`

Defines

`DT_CLOCKS_CTLR_BY_IDX(node_id, idx)`

Get the node identifier for the controller phandle from a “clocks” phandle-array property at an index.

Example devicetree fragment:

```
clk1: clock-controller@... { ... };

clk2: clock-controller@... { ... };

n: node {
        clocks = <&clk1 10 20>, <&clk2 30 40>;
};
```

Example usage:

```
DT_CLOCKS_CTLR_BY_IDX(DT_NODELABEL(n), 0)) // DT_NODELABEL(clk1)
DT_CLOCKS_CTLR_BY_IDX(DT_NODELABEL(n), 1)) // DT_NODELABEL(clk2)
```

See also:

DT_PHANDLE_BY_IDX()

Parameters

- `node_id` – node identifier
- `idx` – logical index into “clocks”

Returns the node identifier for the clock controller referenced at index “idx”

`DT_CLOCKS_CTLR(node_id)`

Equivalent to *DT_CLOCKS_CTLR_BY_IDX(node_id, 0)*

See also:

DT_CLOCKS_CTLR_BY_IDX()

Parameters

- `node_id` – node identifier

Returns a node identifier for the clocks controller at index 0 in “clocks”

`DT_CLOCKS_CTLR_BY_NAME(node_id, name)`

Get the node identifier for the controller phandle from a clocks phandle-array property at an index.

Example devicetree fragment:

```
clk1: clock-controller@... { ... };

clk2: clock-controller@... { ... };

n: node {
        clocks = <&clk1 10 20>, <&clk2 30 40>;
        clock-names = "alpha", "beta";
};
```

Example usage:

```
DT_CLOCKS_CTLR_BY_NAME(DT_NODELABEL(n), beta) // DT_NODELABEL(clk2)
```

See also:

DT_PHANDLE_BY_NAME()

Parameters

- `node_id` – node identifier
- `name` – lowercase-and-underscores name of a clocks element as defined by the node's clock-names property

Returns the node identifier for the clock controller referenced by name

`DT_CLOCKS_LABEL_BY_IDX(node_id, idx)`

Get a label property from the node referenced by a pwms property at an index.

It's an error if the clock controller node referenced by the phandle in node_id's clocks property at index "idx" has no label property.

Example devicetree fragment:

```
clk1: clock-controller@... {
        label = "CLK_1";
};

clk2: clock-controller@... {
        label = "CLK_2";
};

n: node {
        clocks = <&clk1 10 20>, <&clk2 30 40>;
};
```

Example usage:

```
DT_CLOCKS_LABEL_BY_IDX(DT_NODELABEL(n), 1) // "CLK_2"
```

See also:

DT_PROP_BY_PHANDLE_IDX()

Parameters

- `node_id` – node identifier for a node with a clocks property
- `idx` – logical index into clocks property

Returns the label property of the node referenced at index "idx"

`DT_CLOCKS_LABEL_BY_NAME(node_id, name)`

Get a label property from a clocks property by name.

It's an error if the clock controller node referenced by the phandle in node_id's clocks property at the element named "name" has no label property.

Example devicetree fragment:

```
clk1: clock-controller@... {
        label = "CLK_1";
};

clk2: clock-controller@... {
        label = "CLK_2";
};

n: node {
        clocks = <&clk1 10 20>, <&clk2 30 40>;
```

(continues on next page)

(continued from previous page)

```
        clock-names = "alpha", "beta";
};
```

Example usage:

```
DT_CLOCKS_LABEL_BY_NAME(DT_NODELABEL(n), beta) // "CLK_2"
```

See also:

DT_PHANDLE_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a clocks property
- `name` – lowercase-and-underscores name of a clocks element as defined by the node's clock-names property

Returns the label property of the node referenced at the named element

`DT_CLOCKS_LABEL(node_id)`

Equivalent to *DT_CLOCKS_LABEL_BY_IDX(node_id, 0)*

See also:

DT_CLOCKS_LABEL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a clocks property

Returns the label property of the node referenced at index 0

`DT_CLOCKS_CELL_BY_IDX(node_id, idx, cell)`

Get a clock specifier's cell value at an index.

Example devicetree fragment:

```
clk1: clock-controller@... {
        compatible = "vnd,clock";
        #clock-cells = < 2 >;
};

n: node {
        clocks = < &clk1 10 20 >, < &clk1 30 40 >;
};
```

Bindings fragment for the vnd,clock compatible:

```
clock-cells:
  - bus
  - bits
```

Example usage:

```
DT_CLOCKS_CELL_BY_IDX(DT_NODELABEL(n), 0, bus) // 10
DT_CLOCKS_CELL_BY_IDX(DT_NODELABEL(n), 1, bits) // 40
```

See also:

DT_PHA_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a clocks property
- `idx` – logical index into clocks property
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index “idx”

`DT_CLOCKS_CELL_BY_NAME(node_id, name, cell)`

Get a clock specifier’s cell value by name.

Example devicetree fragment:

```
clk1: clock-controller@... {
        compatible = "vnd,clock";
        #clock-cells = < 2 >;
};

n: node {
        clocks = < &clk1 10 20 >, < &clk1 30 40 >;
        clock-names = "alpha", "beta";
};
```

Bindings fragment for the vnd,clock compatible:

```
clock-cells:
  - bus
  - bits
```

Example usage:

```
DT_CLOCKS_CELL_BY_NAME(DT_NODELABEL(n), alpha, bus) // 10
DT_CLOCKS_CELL_BY_NAME(DT_NODELABEL(n), beta, bits) // 40
```

See also:

DT_PHA_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a clocks property
- `name` – lowercase-and-underscores name of a clocks element as defined by the node’s clock-names property
- `cell` – lowercase-and-underscores cell name

Returns the cell value in the specifier at the named element

`DT_CLOCKS_CELL(node_id, cell)`

Equivalent to *DT_CLOCKS_CELL_BY_IDX(node_id, 0, cell)*

See also:

DT_CLOCKS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a clocks property
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index 0

`DT_INST_CLOCKS_CTLR_BY_IDX(inst, idx)`

Get the node identifier for the controller phandle from a "clocks" phandle-array property at an index.

See also:

DT_CLOCKS_CTLR_BY_IDX()

Parameters

- `inst` – instance number
- `idx` – logical index into "clocks"

Returns the node identifier for the clock controller referenced at index "idx"

`DT_INST_CLOCKS_CTLR(inst)`

Equivalent to *DT_INST_CLOCKS_CTLR_BY_IDX(inst, 0)*

See also:

DT_CLOCKS_CTLR()

Parameters

- `inst` – instance number

Returns a node identifier for the clocks controller at index 0 in "clocks"

`DT_INST_CLOCKS_CTLR_BY_NAME(inst, name)`

Get the node identifier for the controller phandle from a clocks phandle-array property by name.

See also:

DT_CLOCKS_CTLR_BY_NAME()

Parameters

- `inst` – instance number
- `name` – lowercase-and-underscores name of a clocks element as defined by the node's clock-names property

Returns the node identifier for the clock controller referenced by the named element

`DT_INST_CLOCKS_LABEL_BY_IDX(inst, idx)`

Get a label property from a DT_DRV_COMPAT instance's clocks property at an index.

See also:

DT_CLOCKS_LABEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into clocks property

Returns the label property of the node referenced at index “idx”

`DT_INST_CLOCKS_LABEL_BY_NAME(inst, name)`

Get a label property from a DT_DRV_COMPAT instance’s clocks property by name.

See also:

DT_CLOCKS_LABEL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a clocks element as defined by the node’s clock-names property

Returns the label property of the node referenced at the named element

`DT_INST_CLOCKS_LABEL(inst)`

Equivalent to *DT_INST_CLOCKS_LABEL_BY_IDX(inst, 0)*

See also:

DT_CLOCKS_LABEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the label property of the node referenced at index 0

`DT_INST_CLOCKS_CELL_BY_IDX(inst, idx, cell)`

Get a DT_DRV_COMPAT instance’s clock specifier’s cell value at an index.

See also:

DT_CLOCKS_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into clocks property
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index “idx”

`DT_INST_CLOCKS_CELL_BY_NAME(inst, name, cell)`

Get a DT_DRV_COMPAT instance’s clock specifier’s cell value by name.

See also:

DT_CLOCKS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a clocks element as defined by the node’s clock-names property

- `cell` – lowercase-and-underscores cell name

Returns the cell value in the specifier at the named element

`DT_INST_CLOCKS_CELL(inst, cell)`

Equivalent to *DT_INST_CLOCKS_CELL_BY_IDX(inst, 0, cell)*

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `cell` – lowercase-and-underscores cell name

Returns the value of the cell inside the specifier at index 0

DMA These conveniences may be used for nodes which describe direct memory access controllers or channels, and properties related to them.

group `devicetree-dmas`

Defines

`DT_DMAS_LABEL_BY_IDX(node_id, idx)`

Get a label property from the node referenced by a dmas property at an index.

It's an error if the DMA controller node referenced by the phandle in node_id's dmas property at index "idx" has no label property.

Example devicetree fragment:

```
dma1: dma@... {
        label = "DMA_1";
};

dma2: dma@... {
        label = "DMA_2";
};

n: node {
    dmas = <&dma1 1 2 0x400 0x3>,
           <&dma2 6 3 0x404 0x5>;
};
```

Example usage:

```
DT_DMAS_LABEL_BY_IDX(DT_NODELABEL(n), 1) // "DMA_2"
```

Parameters

- `node_id` – node identifier for a node with a dmas property
- `idx` – logical index into dmas property

Returns the label property of the node referenced at index "idx"

`DT_INST_DMAS_LABEL_BY_IDX(inst, idx)`

Get a label property from a DT_DRV_COMPAT instance's dmas property at an index.

See also:

DT_DMAS_LABEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into dmas property

Returns the label property of the node referenced at index "idx"

`DT_DMAS_LABEL_BY_NAME(node_id, name)`

Get a label property from a dmas property by name.

It's an error if the DMA controller node referenced by the phandle in node_id's dmas property at the element named "name" has no label property.

Example devicetree fragment:

```
dma1: dma@... {
        label = "DMA_1";
};

dma2: dma@... {
        label = "DMA_2";
};

n: node {
    dmas = <&dma1 1 2 0x400 0x3>,
           <&dma2 6 3 0x404 0x5>;
    dma-names = "tx", "rx";
};
```

Example usage:

```
DT_DMAS_LABEL_BY_NAME(DT_NODELABEL(n), rx) // "DMA_2"
```

Parameters

- `node_id` – node identifier for a node with a dmas property
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns the label property of the node referenced at the named element

`DT_DMAS_CTLR_BY_IDX(node_id, idx)`

Get the node identifier for the DMA controller from a dmas property at an index.

Example devicetree fragment:

```
dma1: dma@... { ... };

dma2: dma@... { ... };

n: node {
    dmas = <&dma1 1 2 0x400 0x3>,
           <&dma2 6 3 0x404 0x5>;
};
```

Example usage:

```
DT_DMAS_CTLR_BY_IDX(DT_NODELABEL(n), 0) // DT_NODELABEL(dma1)
DT_DMAS_CTLR_BY_IDX(DT_NODELABEL(n), 1) // DT_NODELABEL(dma2)
```

See also:

DT_PROP_BY_PHANDLE_IDX()

Parameters

- `node_id` – node identifier for a node with a dmas property
- `idx` – logical index into dmas property

Returns the node identifier for the DMA controller referenced at index "idx"

`DT_DMAS_CTLR_BY_NAME(node_id, name)`

Get the node identifier for the DMA controller from a dmas property by name.

Example devicetree fragment:

```
dma1: dma@... { ... };

dma2: dma@... { ... };

n: node {
    dmas = <&dma1 1 2 0x400 0x3>,
           <&dma2 6 3 0x404 0x5>;
    dma-names = "tx", "rx";
};
```

Example usage:

```
DT_DMAS_CTLR_BY_NAME(DT_NODELABEL(n), tx) // DT_NODELABEL(dma1)
DT_DMAS_CTLR_BY_NAME(DT_NODELABEL(n), rx) // DT_NODELABEL(dma2)
```

See also:

DT_PHANDLE_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a dmas property
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns the node identifier for the DMA controller in the named element

`DT_DMAS_CTLR(node_id)`

Equivalent to *DT_DMAS_CTLR_BY_IDX(node_id, 0)*

See also:

DT_DMAS_CTLR_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a dmas property

Returns the node identifier for the DMA controller at index 0 in the node's "dmas" property

DT_INST_DMAS_LABEL_BY_NAME(inst, name)

Get a label property from a DT_DRV_COMPAT instance's dmas property by name.

See also:

DT_DMAS_LABEL_BY_NAME()

Parameters

- inst – DT_DRV_COMPAT instance number
- name – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns the label property of the node referenced at the named element

DT_INST_DMAS_CTLR_BY_IDX(inst, idx)

Get the node identifier for the DMA controller from a DT_DRV_COMPAT instance's dmas property at an index.

See also:

DT_DMAS_CTLR_BY_IDX()

Parameters

- inst – DT_DRV_COMPAT instance number
- idx – logical index into dmas property

Returns the node identifier for the DMA controller referenced at index "idx"

DT_INST_DMAS_CTLR_BY_NAME(inst, name)

Get the node identifier for the DMA controller from a DT_DRV_COMPAT instance's dmas property by name.

See also:

DT_DMAS_CTLR_BY_NAME()

Parameters

- inst – DT_DRV_COMPAT instance number
- name – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns the node identifier for the DMA controller in the named element

DT_INST_DMAS_CTLR(inst)

Equivalent to *DT_INST_DMAS_CTLR_BY_IDX(inst, 0)*

See also:

DT_DMAS_CTLR_BY_IDX()

Parameters

- inst – DT_DRV_COMPAT instance number

Returns the node identifier for the DMA controller at index 0 in the instance's "dmas" property

DT_DMAS_CELL_BY_IDX(node_id, idx, cell)

Get a DMA specifier's cell value at an index.

Example devicetree fragment:

```
dma1: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

dma2: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

n: node {
    dmas = <&dma1 1 0x400>,
           <&dma2 6 0x404>;
};
```

Bindings fragment for the vnd,dma compatible:

```
dma-cells:
  - channel
  - config
```

Example usage:

```
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 0, channel) // 1
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 1, channel) // 6
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 0, config) // 0x400
DT_DMAS_CELL_BY_IDX(DT_NODELABEL(n), 1, config) // 0x404
```

See also:

DT_PHA_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a dmas property
- `idx` – logical index into dmas property
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index "idx"

DT_INST_DMAS_CELL_BY_IDX(inst, idx, cell)

Get a DT_DRV_COMPAT instance's DMA specifier's cell value at an index.

See also:

DT_DMAS_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into dmas property

- `cell` – lowercase-and-underscores cell name

Returns the cell value at index “idx”

`DT_DMAS_CELL_BY_NAME(node_id, name, cell)`

Get a DMA specifier's cell value by name.

Example devicetree fragment:

```
dma1: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

dma2: dma@... {
        compatible = "vnd,dma";
        #dma-cells = <2>;
};

n: node {
    dmas = <&dma1 1 0x400>,
           <&dma2 6 0x404>;
    dma-names = "tx", "rx";
};
```

Bindings fragment for the vnd,dma compatible:

```
dma-cells:
  - channel
  - config
```

Example usage:

```
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), tx, channel) // 1
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), rx, channel) // 6
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), tx, config) // 0x400
DT_DMAS_CELL_BY_NAME(DT_NODELABEL(n), rx, config) // 0x404
```

See also:

DT_PHA_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a dmas property
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property
- `cell` – lowercase-and-underscores cell name

Returns the cell value in the specifier at the named element

`DT_INST_DMAS_CELL_BY_NAME(inst, name, cell)`

Get a DT_DRV_COMPAT instance's DMA specifier's cell value by name.

See also:

DT_DMAS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property
- `cell` – lowercase-and-underscores cell name

Returns the cell value in the specifier at the named element

`DT_DMAS_HAS_IDX(node_id, idx)`

Is index "idx" valid for a dmas property?

Parameters

- `node_id` – node identifier for a node with a dmas property
- `idx` – logical index into dmas property

Returns 1 if the "dmas" property has index "idx", 0 otherwise

`DT_INST_DMAS_HAS_IDX(inst, idx)`

Is index "idx" valid for a DT_DRV_COMPAT instance's dmas property?

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into dmas property

Returns 1 if the "dmas" property has a specifier at index "idx", 0 otherwise

`DT_DMAS_HAS_NAME(node_id, name)`

Does a dmas property have a named element?

Parameters

- `node_id` – node identifier for a node with a dmas property
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns 1 if the dmas property has the named element, 0 otherwise

`DT_INST_DMAS_HAS_NAME(inst, name)`

Does a DT_DRV_COMPAT instance's dmas property have a named element?

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a dmas element as defined by the node's dma-names property

Returns 1 if the dmas property has the named element, 0 otherwise

Fixed flash partitions These conveniences may be used for the special-purpose `fixed-partitions` compatible used to encode information about flash memory partitions in the device tree. See dts/bindings/mtd/partition.yaml for this compatible's binding.

group `devicetree-fixed-partition`

Defines

DT_NODE_BY_FIXED_PARTITION_LABEL(label)

Get a node identifier for a fixed partition with a given label property.

Example devicetree fragment:

```
flash@... {
        partitions {
                compatible = "fixed-partitions";
                boot_partition: partition@0 {
                        label = "mcuboot";
                };
                slot0_partition: partition@c000 {
                        label = "image-0";
                };
                ...
        };
};
```

Example usage:

```
DT_NODE_BY_FIXED_PARTITION_LABEL(mcuboot) // node identifier for boot_
↪partition
DT_NODE_BY_FIXED_PARTITION_LABEL(image_0) // node identifier for slot0_
↪partition
```

Parameters

- label – lowercase-and-underscores label property value

Returns a node identifier for the partition with that label property

DT_HAS_FIXED_PARTITION_LABEL(label)

Test if a fixed partition with a given label property exists.

Parameters

- label – lowercase-and-underscores label property value

Returns 1 if any "fixed-partitions" child node has the given label, 0 otherwise.

DT_FIXED_PARTITION_ID(node_id)

Get a numeric identifier for a fixed partition.

Parameters

- node_id – node identifier for a fixed-partitions child node

Returns the partition's ID, a unique zero-based index number

DT_MTD_FROM_FIXED_PARTITION(node_id)

Get the node identifier of the flash device for a partition.

Parameters

- node_id – node identifier for a fixed-partitions child node

Returns the node identifier of the memory technology device that contains the fixed-partitions node.

GPIO These conveniences may be used for nodes which describe GPIO controllers/pins, and properties related to them.

group devicetree-gpio

Defines

DT_GPIO_CTLR_BY_IDX(node_id, gpio_pha, idx)

Get the node identifier for the controller phandle from a gpio phandle-array property at an index.

Example devicetree fragment:

```
gpio1: gpio@... { };

gpio2: gpio@... { };

n: node {
        gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                <&gpio2 30 GPIO_ACTIVE_HIGH>;
};
```

Example usage:

```
DT_GPIO_CTLR_BY_IDX(DT_NODELABEL(n), gpios, 1) // DT_NODELABEL(gpio2)
```

See also:

DT_PHANDLE_BY_IDX()

Parameters

- node_id – node identifier
- gpio_pha – lowercase-and-underscores GPIO property with type "phandle-array"
- idx – logical index into "gpio_pha"

Returns the node identifier for the gpio controller referenced at index "idx"

DT_GPIO_CTLR(node_id, gpio_pha)

Equivalent to *DT_GPIO_CTLR_BY_IDX(node_id, gpio_pha, 0)*

See also:

DT_GPIO_CTLR_BY_IDX()

Parameters

- node_id – node identifier
- gpio_pha – lowercase-and-underscores GPIO property with type "phandle-array"

Returns a node identifier for the gpio controller at index 0 in "gpio_pha"

DT_GPIO_LABEL_BY_IDX(node_id, gpio_pha, idx)

Get a label property from a gpio phandle-array property at an index.

It's an error if the GPIO controller node referenced by the phandle in node_id's "gpio_pha" property at index "idx" has no label property.

Example devicetree fragment:

```
gpio1: gpio@... {
        label = "GPIO_1";
};

gpio2: gpio@... {
        label = "GPIO_2";
};

n: node {
        gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                <&gpio2 30 GPIO_ACTIVE_HIGH>;
};
```

Example usage:

```
DT_GPIO_LABEL_BY_IDX(DT_NODELABEL(n), gpios, 1) // "GPIO_2"
```

See also:

DT_PHANDLE_BY_IDX()

Parameters

- `node_id` – node identifier
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"
- `idx` – logical index into "gpio_pha"

Returns the label property of the node referenced at index "idx"

`DT_GPIO_LABEL(node_id, gpio_pha)`

Equivalent to *DT_GPIO_LABEL_BY_IDX(node_id, gpio_pha, 0)*

See also:

DT_GPIO_LABEL_BY_IDX()

Parameters

- `node_id` – node identifier
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"

Returns the label property of the node referenced at index 0

`DT_GPIO_PIN_BY_IDX(node_id, gpio_pha, idx)`

Get a GPIO specifier's pin cell at an index.

This macro only works for GPIO specifiers with cells named "pin". Refer to the node's binding to check if necessary.

Example devicetree fragment:

```
gpio1: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};
```

(continues on next page)

(continued from previous page)

```
gpio2: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};

n: node {
        gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                <&gpio2 30 GPIO_ACTIVE_HIGH>;
};
```

Bindings fragment for the vnd,gpio compatible:

```
gpio-cells:
  - pin
  - flags
```

Example usage:

```
DT_GPIO_PIN_BY_IDX(DT_NODELABEL(n), gpios, 0) // 10
DT_GPIO_PIN_BY_IDX(DT_NODELABEL(n), gpios, 1) // 30
```

See also:

DT_PHA_BY_IDX()

Parameters

- `node_id` – node identifier
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"
- `idx` – logical index into "gpio_pha"

Returns the pin cell value at index "idx"

`DT_GPIO_PIN(node_id, gpio_pha)`

Equivalent to *DT_GPIO_PIN_BY_IDX(node_id, gpio_pha, 0)*

See also:

DT_GPIO_PIN_BY_IDX()

Parameters

- `node_id` – node identifier
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"

Returns the pin cell value at index 0

`DT_GPIO_FLAGS_BY_IDX(node_id, gpio_pha, idx)`

Get a GPIO specifier's flags cell at an index.

This macro expects GPIO specifiers with cells named "flags". If there is no "flags" cell in the GPIO specifier, zero is returned. Refer to the node's binding to check specifier cell names if necessary.

Example devicetree fragment:

```
gpio1: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};

gpio2: gpio@... {
        compatible = "vnd,gpio";
        #gpio-cells = <2>;
};

n: node {
        gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                <&gpio2 30 GPIO_ACTIVE_HIGH>;
};
```

Bindings fragment for the vnd,gpio compatible:

```
gpio-cells:
  - pin
  - flags
```

Example usage:

```
DT_GPIO_FLAGS_BY_IDX(DT_NODELABEL(n), gpios, 0) // GPIO_ACTIVE_LOW
DT_GPIO_FLAGS_BY_IDX(DT_NODELABEL(n), gpios, 1) // GPIO_ACTIVE_HIGH
```

See also:

DT_PHA_BY_IDX()

Parameters

- `node_id` – node identifier
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"
- `idx` – logical index into "gpio_pha"

Returns the flags cell value at index "idx", or zero if there is none

`DT_GPIO_FLAGS(node_id, gpio_pha)`

Equivalent to *DT_GPIO_FLAGS_BY_IDX(node_id, gpio_pha, 0)*

See also:

DT_GPIO_FLAGS_BY_IDX()

Parameters

- `node_id` – node identifier
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"

Returns the flags cell value at index 0, or zero if there is none

`DT_INST_GPIO_LABEL_BY_IDX(inst, gpio_pha, idx)`

Get a label property from a DT_DRV_COMPAT instance's GPIO property at an index.

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"
- `idx` – logical index into "gpio_pha"

Returns the label property of the node referenced at index "idx"

`DT_INST_GPIO_LABEL(inst, gpio_pha)`

Equivalent to *DT_INST_GPIO_LABEL_BY_IDX(inst, gpio_pha, 0)*

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"

Returns the label property of the node referenced at index 0

`DT_INST_GPIO_PIN_BY_IDX(inst, gpio_pha, idx)`

Get a DT_DRV_COMPAT instance's GPIO specifier's pin cell value at an index.

See also:

DT_GPIO_PIN_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"
- `idx` – logical index into "gpio_pha"

Returns the pin cell value at index "idx"

`DT_INST_GPIO_PIN(inst, gpio_pha)`

Equivalent to *DT_INST_GPIO_PIN_BY_IDX(inst, gpio_pha, 0)*

See also:

DT_INST_GPIO_PIN_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"

Returns the pin cell value at index 0

`DT_INST_GPIO_FLAGS_BY_IDX(inst, gpio_pha, idx)`

Get a DT_DRV_COMPAT instance's GPIO specifier's flags cell at an index.

See also:

DT_GPIO_FLAGS_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"
- `idx` – logical index into "gpio_pha"

Returns the flags cell value at index "idx", or zero if there is none

`DT_INST_GPIO_FLAGS(inst, gpio_pha)`

Equivalent to *DT_INST_GPIO_FLAGS_BY_IDX(inst, gpio_pha, 0)*

See also:

DT_INST_GPIO_FLAGS_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `gpio_pha` – lowercase-and-underscores GPIO property with type "phandle-array"

Returns the flags cell value at index 0, or zero if there is none

IO channels These are commonly used by device drivers which need to use IO channels (e.g. ADC or DAC channels) for conversion.

group `devicetree-io-channels`

Defines

`DT_IO_CHANNELS_LABEL_BY_IDX(node_id, idx)`

Get a label property from the node referenced by an io-channels property at an index.

It's an error if the node referenced by the phandle in node_id's io-channels property at index "idx" has no label property.

Example devicetree fragment:

```
adc1: adc@... {
        label = "ADC_1";
};

adc2: adc@... {
        label = "ADC_2";
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
};
```

Example usage:

```
DT_IO_CHANNELS_LABEL_BY_IDX(DT_NODELABEL(n), 1) // "ADC_2"
```

See also:

DT_PROP_BY_PHANDLE_IDX()

Parameters

- `node_id` – node identifier for a node with an io-channels property
- `idx` – logical index into io-channels property

Returns the label property of the node referenced at index “idx”

`DT_IO_CHANNELS_LABEL_BY_NAME(node_id, name)`

Get a label property from an io-channels property by name.

It’s an error if the node referenced by the phandle in node_id’s io-channels property at the element named “name” has no label property.

Example devicetree fragment:

```
adc1: adc@... {
        label = "ADC_1";
};

adc2: adc@... {
        label = "ADC_2";
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Example usage:

```
DT_IO_CHANNELS_LABEL_BY_NAME(DT_NODELABEL(n), bandgap) // "ADC_2"
```

See also:

DT_PHANDLE_BY_NAME()

Parameters

- `node_id` – node identifier for a node with an io-channels property
- `name` – lowercase-and-underscores name of an io-channels element as defined by the node’s io-channel-names property

Returns the label property of the node referenced at the named element

`DT_IO_CHANNELS_LABEL(node_id)`

Equivalent to *DT_IO_CHANNELS_LABEL_BY_IDX(node_id, 0)*

See also:

DT_IO_CHANNELS_LABEL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with an io-channels property

Returns the label property of the node referenced at index 0

`DT_IO_CHANNELS_CTLR_BY_IDX(node_id, idx)`

Get the node identifier for the node referenced by an io-channels property at an index.

Example devicetree fragment:

```
adc1: adc@... { ... };

adc2: adc@... { ... };

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
};
```

Example usage:

```
DT_IO_CHANNELS_CTLR_BY_IDX(DT_NODELABEL(n), 0) // DT_NODELABEL(adc1)
DT_IO_CHANNELS_CTLR_BY_IDX(DT_NODELABEL(n), 1) // DT_NODELABEL(adc2)
```

See also:

DT_PROP_BY_PHANDLE_IDX()

Parameters

- `node_id` – node identifier for a node with an io-channels property
- `idx` – logical index into io-channels property

Returns the node identifier for the node referenced at index "idx"

`DT_IO_CHANNELS_CTLR_BY_NAME(node_id, name)`

Get the node identifier for the node referenced by an io-channels property by name.

Example devicetree fragment:

```
adc1: adc@... { ... };

adc2: adc@... { ... };

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Example usage:

DT_IO_CHANNELS_CTLR_BY_NAME(DT_NODELABEL(n), sensor) // *DT_NODELABEL(adc1)*
DT_IO_CHANNELS_CTLR_BY_NAME(DT_NODELABEL(n), bandgap) // *DT_NODELABEL(adc2)*

See also:

DT_PHANDLE_BY_NAME()

Parameters

- `node_id` – node identifier for a node with an io-channels property
- `name` – lowercase-and-underscores name of an io-channels element as defined by the node's io-channel-names property

Returns the node identifier for the node referenced at the named element

`DT_IO_CHANNELS_CTLR(node_id)`

Equivalent to *DT_IO_CHANNELS_CTLR_BY_IDX(node_id, 0)*

See also:

DT_IO_CHANNELS_CTLR_BY_IDX()

Parameters

- `node_id` – node identifier for a node with an io-channels property

Returns the node identifier for the node referenced at index 0 in the node's "io-channels" property

`DT_INST_IO_CHANNELS_LABEL_BY_IDX(inst, idx)`

Get a label property from a DT_DRV_COMPAT instance's io-channels property at an index.

See also:

DT_IO_CHANNELS_LABEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into io-channels property

Returns the label property of the node referenced at index "idx"

`DT_INST_IO_CHANNELS_LABEL_BY_NAME(inst, name)`

Get a label property from a DT_DRV_COMPAT instance's io-channels property by name.

See also:

DT_IO_CHANNELS_LABEL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of an io-channels element as defined by the instance's io-channel-names property

Returns the label property of the node referenced at the named element

`DT_INST_IO_CHANNELS_LABEL(inst)`

Equivalent to *DT_INST_IO_CHANNELS_LABEL_BY_IDX(inst, 0)*

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the label property of the node referenced at index 0

`DT_INST_IO_CHANNELS_CTLR_BY_IDX(inst, idx)`

Get the node identifier from a DT_DRV_COMPAT instance's io-channels property at an index.

See also:

DT_IO_CHANNELS_CTLR_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into io-channels property

Returns the node identifier for the node referenced at index “idx”

`DT_INST_IO_CHANNELS_CTLR_BY_NAME(inst, name)`

Get the node identifier from a DT_DRV_COMPAT instance’s io-channels property by name.

See also:

DT_IO_CHANNELS_CTLR_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of an io-channels element as defined by the node’s io-channel-names property

Returns the node identifier for the node referenced at the named element

`DT_INST_IO_CHANNELS_CTLR(inst)`

Equivalent to *DT_INST_IO_CHANNELS_CTLR_BY_IDX(inst, 0)*

See also:

DT_IO_CHANNELS_CTLR_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the node identifier for the node referenced at index 0 in the node’s “io-channels” property

`DT_IO_CHANNELS_INPUT_BY_IDX(node_id, idx)`

Get an io-channels specifier input cell at an index.

This macro only works for io-channels specifiers with cells named “input”. Refer to the node’s binding to check if necessary.

Example devicetree fragment:

```
adc1: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

adc2: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
};
```

Bindings fragment for the vnd,adc compatible:

io-channel-cells:

- input

Example usage:

```
DT_IO_CHANNELS_INPUT_BY_IDX(DT_NODELABEL(n), 0) // 10
DT_IO_CHANNELS_INPUT_BY_IDX(DT_NODELABEL(n), 1) // 20
```

See also:

DT_PHA_BY_IDX()

Parameters

- `node_id` – node identifier for a node with an io-channels property
- `idx` – logical index into io-channels property

Returns the input cell in the specifier at index "idx"

`DT_IO_CHANNELS_INPUT_BY_NAME(node_id, name)`

Get an io-channels specifier input cell by name.

This macro only works for io-channels specifiers with cells named "input". Refer to the node's binding to check if necessary.

Example devicetree fragment:

```
adc1: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

adc2: adc@... {
        compatible = "vnd,adc";
        #io-channel-cells = <1>;
};

n: node {
        io-channels = <&adc1 10>, <&adc2 20>;
        io-channel-names = "SENSOR", "BANDGAP";
};
```

Bindings fragment for the vnd,adc compatible:

io-channel-cells:

- input

Example usage:

```
DT_IO_CHANNELS_INPUT_BY_NAME(DT_NODELABEL(n), sensor) // 10
DT_IO_CHANNELS_INPUT_BY_NAME(DT_NODELABEL(n), bandgap) // 20
```

See also:

DT_PHA_BY_NAME()

Parameters

- `node_id` – node identifier for a node with an io-channels property
- `name` – lowercase-and-underscores name of an io-channels element as defined by the node's io-channel-names property

Returns the input cell in the specifier at the named element

DT_IO_CHANNELS_INPUT(node_id)

Equivalent to *DT_IO_CHANNELS_INPUT_BY_IDX(node_id, 0)*

See also:

DT_IO_CHANNELS_INPUT_BY_IDX()

Parameters

- node_id – node identifier for a node with an io-channels property

Returns the input cell in the specifier at index 0

DT_INST_IO_CHANNELS_INPUT_BY_IDX(inst, idx)

Get an input cell from the "DT_DRV_INST(inst)" io-channels property at an index.

See also:

DT_IO_CHANNELS_INPUT_BY_IDX()

Parameters

- inst – DT_DRV_COMPAT instance number
- idx – logical index into io-channels property

Returns the input cell in the specifier at index "idx"

DT_INST_IO_CHANNELS_INPUT_BY_NAME(inst, name)

Get an input cell from the "DT_DRV_INST(inst)" io-channels property by name.

See also:

DT_IO_CHANNELS_INPUT_BY_NAME()

Parameters

- inst – DT_DRV_COMPAT instance number
- name – lowercase-and-underscores name of an io-channels element as defined by the instance's io-channel-names property

Returns the input cell in the specifier at the named element

DT_INST_IO_CHANNELS_INPUT(inst)

Equivalent to *DT_INST_IO_CHANNELS_INPUT_BY_IDX(inst, 0)*

Parameters

- inst – DT_DRV_COMPAT instance number

Returns the input cell in the specifier at index 0

Pinctrl (pin control) These are used to access pin control properties by name or index.

Devicetree nodes may have properties which specify pin control (sometimes known as pin mux) settings. These are expressed using pinctrl-<index> properties within the node, where the <index> values are contiguous integers starting from 0. These may also be named using the pinctrl-names property.

Here is an example:

```
node {
    ...
    pinctrl-0 = <&foo &bar ...>;
    pinctrl-1 = <&baz ...>;
    pinctrl-names = "default", "sleep";
};
```

Above, `pinctrl-0` has name `"default"`, and `pinctrl-1` has name `"sleep"`. The `pinctrl-<index>` property values contain phandles. The `&foo`, `&bar`, etc. phandles within the properties point to nodes whose contents vary by platform, and which describe a pin configuration for the node.

group `devicetree-pinctrl`

Defines

`DT_PINCTRL_BY_IDX(node_id, pc_idx, idx)`

Get a node identifier for a phandle in a pinctrl property by index.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz &blub>;
}
```

Example usage:

```
DT_PINCTRL_BY_IDX(DT_NODELABEL(n), 0, 1) // DT_NODELABEL(bar)
DT_PINCTRL_BY_IDX(DT_NODELABEL(n), 1, 0) // DT_NODELABEL(baz)
```

Parameters

- `node_id` – node with a pinctrl-‘pc_idx’ property
- `pc_idx` – index of the pinctrl property itself
- `idx` – index into the value of the pinctrl property

Returns node identifier for the phandle at index ‘idx’ in ‘pinctrl-‘pc_idx”

`DT_PINCTRL_0(node_id, idx)`

Get a node identifier from a pinctrl-0 property.

This is equivalent to:

```
DT_PINCTRL_BY_IDX(node_id, 0, idx)
```

It is provided for convenience since pinctrl-0 is commonly used.

Parameters

- `node_id` – node with a pinctrl-0 property
- `idx` – index into the pinctrl-0 property

Returns node identifier for the phandle at index idx in the pinctrl-0 property of that node

`DT_PINCTRL_BY_NAME(node_id, name, idx)`

Get a node identifier for a phandle inside a pinctrl node by name.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz &blub>;
        pinctrl-names = "default", "sleep";
};
```

Example usage:

```
DT_PINCTRL_BY_NAME(DT_NODELABEL(n), default, 1) // DT_NODELABEL(bar)
DT_PINCTRL_BY_NAME(DT_NODELABEL(n), sleep, 0) // DT_NODELABEL(baz)
```

Parameters

- `node_id` – node with a named pinctrl property
- `name` – lowercase-and-underscores pinctrl property name
- `idx` – index into the value of the named pinctrl property

Returns node identifier for the phandle at that index in the pinctrl property

`DT_PINCTRL_NAME_TO_IDX(node_id, name)`

Convert a pinctrl name to its corresponding index.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz &blub>;
        pinctrl-names = "default", "sleep";
};
```

Example usage:

```
DT_PINCTRL_NAME_TO_IDX(DT_NODELABEL(n), default) // 0
DT_PINCTRL_NAME_TO_IDX(DT_NODELABEL(n), sleep)   // 1
```

Parameters

- `node_id` – node identifier with a named pinctrl property
- `name` – lowercase-and-underscores name name of the pinctrl whose index to get

Returns integer literal for the index of the pinctrl property with that name

`DT_PINCTRL_IDX_TO_NAME_TOKEN(node_id, pc_idx)`

Convert a pinctrl property index to its name as a token.

This allows you to get a pinctrl property's name, and "remove the

quotes" from it.

DT_PINCTRL_IDX_TO_NAME_TOKEN() can only be used if the node has a pinctrl-'pc_idx' property and a pinctrl-names property element for that index. It is an error to use it in other circumstances.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
        pinctrl-names = "default", "f.o.o2";
};
```

Example usage:

```
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 0) // default
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 1) // f_o_o2
```

The same caveats and restrictions that apply to *DT_STRING_TOKEN()*'s return value also apply here.

Parameters

- `node_id` – node identifier
- `pc_idx` – index of a pinctrl property in that node

Returns name of the pinctrl property, as a token, without any quotes and with non-alphanumeric characters converted to underscores

`DT_PINCTRL_IDX_TO_NAME_UPPER_TOKEN(node_id, pc_idx)`

Like *DT_PINCTRL_IDX_TO_NAME_TOKEN()*, but with an uppercased result.

This does the a similar conversion as *DT_PINCTRL_IDX_TO_NAME_TOKEN(node_id, pc_idx)*. The only difference is that alphabetical characters in the result are uppercased.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
        pinctrl-names = "default", "f.o.o2";
};
```

Example usage:

```
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 0) // DEFAULT
DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_NODELABEL(n), 1) // F_O_O2
```

The same caveats and restrictions that apply to *DT_STRING_UPPER_TOKEN()*'s return value also apply here.

`DT_NUM_PINCTRLS_BY_IDX(node_id, pc_idx)`

Get the number of phandles in a pinctrl property.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <&foo &bar>;
};

n2: node-2 {
        pinctrl-0 = <&baz>;
};
```

Example usage:

```
DT_NUM_PINCTRLS_BY_IDX(DT_NODELABEL(n1), 0) // 2
DT_NUM_PINCTRLS_BY_IDX(DT_NODELABEL(n2), 0) // 1
```

Parameters

- `node_id` – node identifier with a pinctrl property
- `pc_idx` – index of the pinctrl property itself

Returns number of phandles in the property with that index

`DT_NUM_PINCTRLS_BY_NAME(node_id, name)`

Like *DT_NUM_PINCTRLS_BY_IDX()*, but by name instead.

Example devicetree fragment:

```
n: node {
        pinctrl-0 = <&foo &bar>;
        pinctrl-1 = <&baz>
        pinctrl-names = "default", "sleep";
};
```

Example usage:

```
DT_NUM_PINCTRLS_BY_NAME(DT_NODELABEL(n), default) // 2
DT_NUM_PINCTRLS_BY_NAME(DT_NODELABEL(n), sleep)   // 1
```

Parameters

- `node_id` – node identifier with a pinctrl property
- `name` – lowercase-and-underscores name name of the pinctrl property

Returns number of phandles in the property with that name

`DT_NUM_PINCTRL_STATES(node_id)`

Get the number of pinctrl properties in a node.

This expands to 0 if there are no pinctrl-i properties. Otherwise, it expands to the number of such properties.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
};

n2: node-2 {
};
```

Example usage:

```
DT_NUM_PINCTRL_STATES(DT_NODELABEL(n1)) // 2
DT_NUM_PINCTRL_STATES(DT_NODELABEL(n2)) // 0
```

Parameters

- `node_id` – node identifier; may or may not have any pinctrl properties

Returns number of pinctrl properties in the node

`DT_PINCTRL_HAS_IDX(node_id, pc_idx)`

Test if a node has a pinctrl property with an index.

This expands to 1 if the pinctrl-'idx' property exists. Otherwise, it expands to 0.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <...>;
        pinctrl-1 = <...>;
};

n2: node-2 {
};
```

Example usage:

```
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n1), 0) // 1
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n1), 1) // 1
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n1), 2) // 0
DT_PINCTRL_HAS_IDX(DT_NODELABEL(n2), 0) // 0
```

Parameters

- `node_id` – node identifier; may or may not have any pinctrl properties
- `pc_idx` – index of a pinctrl property whose existence to check

Returns 1 if the property exists, 0 otherwise

`DT_PINCTRL_HAS_NAME(node_id, name)`

Test if a node has a pinctrl property with a name.

This expands to 1 if the named pinctrl property exists. Otherwise, it expands to 0.

Example devicetree fragment:

```
n1: node-1 {
        pinctrl-0 = <...>;
        pinctrl-names = "default";
};

n2: node-2 {
};
```

Example usage:

```
DT_PINCTRL_HAS_NAME(DT_NODELABEL(n1), default) // 1
DT_PINCTRL_HAS_NAME(DT_NODELABEL(n1), sleep)   // 0
DT_PINCTRL_HAS_NAME(DT_NODELABEL(n2), default) // 0
```

Parameters

- `node_id` – node identifier; may or may not have any pinctrl properties
- `name` – lowercase-and-underscores pinctrl property name to check

Returns 1 if the property exists, 0 otherwise

`DT_INST_PINCTRL_BY_IDX(inst, pc_idx, idx)`

Get a node identifier for a phandle in a pinctrl property by index for a DT_DRV_COMPAT instance.

This is equivalent to *DT_PINCTRL_BY_IDX(DT_DRV_INST(inst), pc_idx, idx)*.

Parameters

- `inst` – instance number

- pc_idx – index of the pinctrl property itself
- idx – index into the value of the pinctrl property

Returns node identifier for the phandle at index 'idx' in 'pinctrl-'pc_idx"

DT_INST_PINCTRL_0(inst, idx)

Get a node identifier from a pinctrl-0 property for a DT_DRV_COMPAT instance.

This is equivalent to:

```
DT_PINCTRL_BY_IDX(DT_DRV_INST(inst), 0, idx)
```

It is provided for convenience since pinctrl-0 is commonly used.

Parameters

- inst – instance number
- idx – index into the pinctrl-0 property

Returns node identifier for the phandle at index idx in the pinctrl-0 property of that instance

DT_INST_PINCTRL_BY_NAME(inst, name, idx)

Get a node identifier for a phandle inside a pinctrl node for a DT_DRV_COMPAT instance.

This is equivalent to *DT_PINCTRL_BY_NAME(DT_DRV_INST(inst), name, idx)*.

Parameters

- inst – instance number
- name – lowercase-and-underscores pinctrl property name
- idx – index into the value of the named pinctrl property

Returns node identifier for the phandle at that index in the pinctrl property

DT_INST_PINCTRL_NAME_TO_IDX(inst, name)

Convert a pinctrl name to its corresponding index for a DT_DRV_COMPAT instance.

This is equivalent to *DT_PINCTRL_NAME_TO_IDX(DT_DRV_INST(inst),name)*.

Parameters

- inst – instance number
- name – lowercase-and-underscores name of the pinctrl whose index to get

Returns integer literal for the index of the pinctrl property with that name

DT_INST_PINCTRL_IDX_TO_NAME_TOKEN(inst, pc_idx)

Convert a pinctrl index to its name as an uppercased token.

This is equivalent to *DT_PINCTRL_IDX_TO_NAME_TOKEN(DT_DRV_INST(inst), pc_idx)*.

Parameters

- inst – instance number
- pc_idx – index of the pinctrl property itself

Returns name of the pin control property as a token

DT_INST_PINCTRL_IDX_TO_NAME_UPPER_TOKEN(inst, pc_idx)

Convert a pinctrl index to its name as an uppercased token.

This is equivalent to *DT_PINCTRL_IDX_TO_NAME_UPPER_TOKEN(DT_DRV_INST(inst), idx)*.

Parameters

- inst – instance number

- pc_idx – index of the pinctrl property itself

Returns name of the pin control property as an uppercase token

DT_INST_NUM_PINCTRLS_BY_IDX(inst, pc_idx)

Get the number of phandles in a pinctrl property for a DT_DRV_COMPAT instance.

This is equivalent to *DT_NUM_PINCTRLS_BY_IDX(DT_DRV_INST(inst),pc_idx)*.

Parameters

- inst – instance number
- pc_idx – index of the pinctrl property itself

Returns number of phandles in the property with that index

DT_INST_NUM_PINCTRLS_BY_NAME(inst, name)

Like *DT_INST_NUM_PINCTRLS_BY_IDX()*, but by name instead.

This is equivalent to *DT_NUM_PINCTRLS_BY_NAME(DT_DRV_INST(inst),name)*.

Parameters

- inst – instance number
- name – lowercase-and-underscores name of the pinctrl property

Returns number of phandles in the property with that name

DT_INST_NUM_PINCTRL_STATES(inst)

Get the number of pinctrl properties in a DT_DRV_COMPAT instance.

This is equivalent to *DT_NUM_PINCTRL_STATES(DT_DRV_INST(inst))*.

Parameters

- inst – instance number

Returns number of pinctrl properties in the instance

DT_INST_PINCTRL_HAS_IDX(inst, pc_idx)

Test if a DT_DRV_COMPAT instance has a pinctrl property with an index.

This is equivalent to *DT_PINCTRL_HAS_IDX(DT_DRV_INST(inst), pc_idx)*.

Parameters

- inst – instance number
- pc_idx – index of a pinctrl property whose existence to check

Returns 1 if the property exists, 0 otherwise

DT_INST_PINCTRL_HAS_NAME(inst, name)

Test if a DT_DRV_COMPAT instance has a pinctrl property with a name.

This is equivalent to *DT_PINCTRL_HAS_NAME(DT_DRV_INST(inst), name)*.

Parameters

- inst – instance number
- name – lowercase-and-underscores pinctrl property name to check

Returns 1 if the property exists, 0 otherwise

PWM These conveniences may be used for nodes which describe PWM controllers and properties related to them.

group devicetree-pwms

Defines

DT_PWMS_LABEL_BY_IDX(node_id, idx)

Get a label property from a pwms property at an index.

It's an error if the PWM controller node referenced by the phandle in node_id's pwms property at index "idx" has no label property.

Example devicetree fragment:

```
pwm1: pwm-controller@... {
        label = "PWM_1";
};

pwm2: pwm-controller@... {
        label = "PWM_2";
};

n: node {
        pwms = <&pwm1 1 PWM_POLARITY_NORMAL>,
               <&pwm2 3 PWM_POLARITY_INVERTED>;
};
```

Example usage:

```
DT_PWMS_LABEL_BY_IDX(DT_NODELABEL(n), 0) // "PWM_1"
DT_PWMS_LABEL_BY_IDX(DT_NODELABEL(n), 1) // "PWM_2"
```

See also:

DT_PROP_BY_PHANDLE_IDX()

Parameters

- node_id – node identifier for a node with a pwms property
- idx – logical index into pwms property

Returns the label property of the node referenced at index "idx"

DT_PWMS_LABEL_BY_NAME(node_id, name)

Get a label property from a pwms property by name.

It's an error if the PWM controller node referenced by the phandle in node_id's pwms property at the element named "name" has no label property.

Example devicetree fragment:

```
pwm1: pwm-controller@... {
        label = "PWM_1";
};

pwm2: pwm-controller@... {
        label = "PWM_2";
};

n: node {
        pwms = <&pwm1 1 PWM_POLARITY_NORMAL>,
               <&pwm2 3 PWM_POLARITY_INVERTED>;
        pwm-names = "alpha", "beta";
};
```

Example usage:

```
DT_PWMS_LABEL_BY_NAME(DT_NODELABEL(n), alpha) // "PWM_1"
DT_PWMS_LABEL_BY_NAME(DT_NODELABEL(n), beta)  // "PWM_2"
```

See also:

DT_PHANDLE_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns the label property of the node referenced at the named element

`DT_PWMS_LABEL(node_id)`

Equivalent to *DT_PWMS_LABEL_BY_IDX(node_id, 0)*

See also:

DT_PWMS_LABEL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property

Returns the label property of the node referenced at index 0

`DT_PWMS_CTLR_BY_IDX(node_id, idx)`

Get the node identifier for the PWM controller from a pwms property at an index.

Example devicetree fragment:

```
pwm1: pwm-controller@... { ... };

pwm2: pwm-controller@... { ... };

n: node {
        pwms = <&pwm1 1 PWM_POLARITY_NORMAL>,
               <&pwm2 3 PWM_POLARITY_INVERTED>;
};
```

Example usage:

```
DT_PWMS_CTLR_BY_IDX(DT_NODELABEL(n), 0) // DT_NODELABEL(pwm1)
DT_PWMS_CTLR_BY_IDX(DT_NODELABEL(n), 1) // DT_NODELABEL(pwm2)
```

See also:

DT_PROP_BY_PHANDLE_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property

Returns the node identifier for the PWM controller referenced at index "idx"

DT_PWMS_CTLR_BY_NAME(node_id, name)

Get the node identifier for the PWM controller from a pwms property by name.

Example devicetree fragment:

```
pwm1: pwm-controller@... { ... };
```

pwm2: pwm-controller... { ... };

n: node { pwms = <&pwm1 1 PWM_POLARITY_NORMAL>, <&pwm2 3 PWM_POLARITY_INVERTED>; pwm-names = "alpha", "beta"; };

Example usage:

```
DT_PWMS_CTLR_BY_NAME(DT_NODELABEL(n), alpha) // DT_NODELABEL(pwm1)
DT_PWMS_CTLR_BY_NAME(DT_NODELABEL(n), beta)  // DT_NODELABEL(pwm2)
```

See also:

DT_PHANDLE_BY_NAME()

Parameters

- node_id – node identifier for a node with a pwms property
- name – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns the node identifier for the PWM controller in the named element

DT_PWMS_CTLR(node_id)

Equivalent to *DT_PWMS_CTLR_BY_IDX(node_id, 0)*

See also:

DT_PWMS_CTLR_BY_IDX()

Parameters

- node_id – node identifier for a node with a pwms property

Returns the node identifier for the PWM controller at index 0 in the node's "pwms" property

DT_PWMS_CELL_BY_IDX(node_id, idx, cell)

Get PWM specifier's cell value at an index.

Example devicetree fragment:

```
pwm1: pwm-controller@... {
        compatible = "vnd,pwm";
        label = "PWM_1";
        #pwm-cells = <2>;
};

pwm2: pwm-controller@... {
        compatible = "vnd,pwm";
        label = "PWM_2";
        #pwm-cells = <2>;
};

n: node {
```

(continues on next page)

(continued from previous page)

```
        pwms = <&pwm1 1 200000 PWM_POLARITY_NORMAL>,
               <&pwm2 3 100000 PWM_POLARITY_INVERTED>;
};
```

Bindings fragment for the “vnd,pwm” compatible:

```
pwm-cells:
  - channel
  - period
  - flags
```

Example usage:

```
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 0, channel) // 1
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 1, channel) // 3
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 0, period)  // 200000
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 1, period)  // 100000
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 0, flags)   // PWM_POLARITY_NORMAL
DT_PWMS_CELL_BY_IDX(DT_NODELABEL(n), 1, flags)   // PWM_POLARITY_INVERTED
```

See also:

DT_PHA_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index “idx”

`DT_PWMS_CELL_BY_NAME`(node_id, name, cell)

Get a PWM specifier’s cell value by name.

Example devicetree fragment:

```
pwm1: pwm-controller@... {
        compatible = "vnd,pwm";
        label = "PWM_1";
        #pwm-cells = <2>;
};

pwm2: pwm-controller@... {
        compatible = "vnd,pwm";
        label = "PWM_2";
        #pwm-cells = <2>;
};

n: node {
        pwms = <&pwm1 1 200000 PWM_POLARITY_NORMAL>,
               <&pwm2 3 100000 PWM_POLARITY_INVERTED>;
        pwm-names = "alpha", "beta";
};
```

Bindings fragment for the “vnd,pwm” compatible:

```
pwm-cells:
  - channel
  - period
  - flags
```

Example usage:

```
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), alpha, channel) // 1
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), beta, channel)  // 3
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), alpha, period)  // 200000
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), beta, period)   // 100000
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), alpha, flags)   // PWM_POLARITY_NORMAL
DT_PWMS_CELL_BY_NAME(DT_NODELABEL(n), beta, flags)    // PWM_POLARITY_
↪INVERTED
```

See also:

DT_PHA_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property
- `cell` – lowercase-and-underscores cell name

Returns the cell value in the specifier at the named element

`DT_PWMS_CELL(node_id, cell)`

Equivalent to *DT_PWMS_CELL_BY_IDX(node_id, 0, cell)*

See also:

DT_PWMS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index 0

`DT_PWMS_CHANNEL_BY_IDX(node_id, idx)`

Get a PWM specifier's channel cell value at an index.

This macro only works for PWM specifiers with cells named "channel". Refer to the node's binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_IDX(node_id, idx, channel)*.

See also:

DT_PWMS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property

Returns the channel cell value at index “idx”

`DT_PWMS_CHANNEL_BY_NAME(node_id, name)`

Get a PWM specifier’s channel cell value by name.

This macro only works for PWM specifiers with cells named “channel”. Refer to the node’s binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_NAME(node_id, name, channel)*.

See also:

DT_PWMS_CELL_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node’s pwm-names property

Returns the channel cell value in the specifier at the named element

`DT_PWMS_CHANNEL(node_id)`

Equivalent to *DT_PWMS_CHANNEL_BY_IDX(node_id, 0)*

See also:

DT_PWMS_CHANNEL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property

Returns the channel cell value at index 0

`DT_PWMS_PERIOD_BY_IDX(node_id, idx)`

Get PWM specifier’s period cell value at an index.

This macro only works for PWM specifiers with cells named “period”. Refer to the node’s binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_IDX(node_id, idx, period)*.

See also:

DT_PWMS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property

Returns the period cell value at index “idx”

`DT_PWMS_PERIOD_BY_NAME(node_id, name)`

Get a PWM specifier’s period cell value by name.

This macro only works for PWM specifiers with cells named “period”. Refer to the node’s binding to check if necessary.

This is equivalent to *DT_PWMS_CELL_BY_NAME(node_id, name, period)*.

See also:

DT_PWMS_CELL_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns the period cell value in the specifier at the named element

`DT_PWMS_PERIOD(node_id)`

Equivalent to *DT_PWMS_PERIOD_BY_IDX(node_id, 0)*

See also:

DT_PWMS_PERIOD_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property

Returns the period cell value at index 0

`DT_PWMS_FLAGS_BY_IDX(node_id, idx)`

Get a PWM specifier's flags cell value at an index.

This macro expects PWM specifiers with cells named "flags". If there is no "flags" cell in the PWM specifier, zero is returned. Refer to the node's binding to check specifier cell names if necessary.

This is equivalent to *DT_PWMS_CELL_BY_IDX(node_id, idx, flags)*.

See also:

DT_PWMS_CELL_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `idx` – logical index into pwms property

Returns the flags cell value at index "idx", or zero if there is none

`DT_PWMS_FLAGS_BY_NAME(node_id, name)`

Get a PWM specifier's flags cell value by name.

This macro expects PWM specifiers with cells named "flags". If there is no "flags" cell in the PWM specifier, zero is returned. Refer to the node's binding to check specifier cell names if necessary.

This is equivalent to *DT_PWMS_CELL_BY_NAME(node_id, name, flags)* if there is a flags cell, but expands to zero if there is none.

See also:

DT_PWMS_CELL_BY_NAME()

Parameters

- `node_id` – node identifier for a node with a pwms property
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns the flags cell value in the specifier at the named element, or zero if there is none

`DT_PWMS_FLAGS(node_id)`

Equivalent to *DT_PWMS_FLAGS_BY_IDX(node_id, 0)*

See also:

DT_PWMS_FLAGS_BY_IDX()

Parameters

- `node_id` – node identifier for a node with a pwms property

Returns the flags cell value at index 0, or zero if there is none

`DT_INST_PWMS_LABEL_BY_IDX(inst, idx)`

Get a label property from a DT_DRV_COMPAT instance's pwms property by name.

See also:

DT_PWMS_LABEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns the label property of the node referenced at index "idx"

`DT_INST_PWMS_LABEL_BY_NAME(inst, name)`

Get a label property from a DT_DRV_COMPAT instance's pwms property by name.

See also:

DT_PWMS_LABEL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns the label property of the node referenced at the named element

`DT_INST_PWMS_LABEL(inst)`

Equivalent to *DT_INST_PWMS_LABEL_BY_IDX(inst, 0)*

See also:

DT_PWMS_LABEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the label property of the node referenced at index 0

`DT_INST_PWMS_CTLR_BY_IDX(inst, idx)`

Get the node identifier for the PWM controller from a DT_DRV_COMPAT instance's pwms property at an index.

See also:

DT_PWMS_CTLR_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns the node identifier for the PWM controller referenced at index "idx"

`DT_INST_PWMS_CTLR_BY_NAME(inst, name)`

Get the node identifier for the PWM controller from a DT_DRV_COMPAT instance's pwms property by name.

See also:

DT_PWMS_CTLR_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns the node identifier for the PWM controller in the named element

`DT_INST_PWMS_CTLR(inst)`

Equivalent to *DT_INST_PWMS_CTLR_BY_IDX(inst, 0)*

See also:

DT_PWMS_CTLR_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the node identifier for the PWM controller at index 0 in the instance's "pwms" property

`DT_INST_PWMS_CELL_BY_IDX(inst, idx, cell)`

Get a DT_DRV_COMPAT instance's PWM specifier's cell value at an index.

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index “idx”

`DT_INST_PWMS_CELL_BY_NAME`(inst, name, cell)

Get a DT_DRV_COMPAT instance’s PWM specifier’s cell value by name.

See also:

DT_PWMS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node’s pwm-names property
- `cell` – lowercase-and-underscores cell name

Returns the cell value in the specifier at the named element

`DT_INST_PWMS_CELL`(inst, cell)

Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, 0, cell)*

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `cell` – lowercase-and-underscores cell name

Returns the cell value at index 0

`DT_INST_PWMS_CHANNEL_BY_IDX`(inst, idx)

Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, idx, channel)*

See also:

DT_INST_PWMS_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns the channel cell value at index “idx”

`DT_INST_PWMS_CHANNEL_BY_NAME`(inst, name)

Equivalent to *DT_INST_PWMS_CELL_BY_NAME(inst, name, channel)*

See also:

DT_INST_PWMS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node’s pwm-names property

Returns the channel cell value in the specifier at the named element

DT_INST_PWMS_CHANNEL(inst)

Equivalent to *DT_INST_PWMS_CHANNEL_BY_IDX(inst, 0)*

See also:

DT_INST_PWMS_CHANNEL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the channel cell value at index 0

DT_INST_PWMS_PERIOD_BY_IDX(inst, idx)

Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, idx, period)*

See also:

DT_INST_PWMS_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns the period cell value at index “idx”

DT_INST_PWMS_PERIOD_BY_NAME(inst, name)

Equivalent to *DT_INST_PWMS_CELL_BY_NAME(inst, name, period)*

See also:

DT_INST_PWMS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node’s pwm-names property

Returns the period cell value in the specifier at the named element

DT_INST_PWMS_PERIOD(inst)

Equivalent to *DT_INST_PWMS_PERIOD_BY_IDX(inst, 0)*

See also:

DT_INST_PWMS_PERIOD_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the period cell value at index 0

`DT_INST_PWMS_FLAGS_BY_IDX(inst, idx)`

Equivalent to *DT_INST_PWMS_CELL_BY_IDX(inst, idx, flags)*

See also:

DT_INST_PWMS_CELL_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `idx` – logical index into pwms property

Returns the flags cell value at index "idx", or zero if there is none

`DT_INST_PWMS_FLAGS_BY_NAME(inst, name)`

Equivalent to *DT_INST_PWMS_CELL_BY_NAME(inst, name, flags)*

See also:

DT_INST_PWMS_CELL_BY_NAME()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `name` – lowercase-and-underscores name of a pwms element as defined by the node's pwm-names property

Returns the flags cell value in the specifier at the named element, or zero if there is none

`DT_INST_PWMS_FLAGS(inst)`

Equivalent to *DT_INST_PWMS_FLAGS_BY_IDX(inst, 0)*

See also:

DT_INST_PWMS_FLAGS_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns the flags cell value at index 0, or zero if there is none

SPI These conveniences may be used for nodes which describe either SPI controllers or devices, depending on the case.

group `devicetree-spi`

Defines

`DT_SPI_HAS_CS_GPIOS(spi)`

Does a SPI controller node have chip select GPIOs configured?

SPI bus controllers use the "cs-gpios" property for configuring chip select GPIOs. Its value is a phandle-array which specifies the chip select lines.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;
};

spi2: spi@... {
        compatible = "vnd,spi";
};
```

Example usage:

```
DT_SPI_HAS_CS_GPIOS(DT_NODELABEL(spi1)) // 1
DT_SPI_HAS_CS_GPIOS(DT_NODELABEL(spi2)) // 0
```

Parameters

- `spi` – a SPI bus controller node identifier

Returns 1 if "spi" has a cs-gpios property, 0 otherwise

`DT_SPI_NUM_CS_GPIOS(spi)`

Number of chip select GPIOs in a SPI controller's cs-gpios property.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;
};

spi2: spi@... {
        compatible = "vnd,spi";
};
```

Example usage:

```
DT_SPI_NUM_CS_GPIOS(DT_NODELABEL(spi1)) // 2
DT_SPI_NUM_CS_GPIOS(DT_NODELABEL(spi2)) // 0
```

Parameters

- `spi` – a SPI bus controller node identifier

Returns Logical length of spi's cs-gpios property, or 0 if "spi" doesn't have a cs-gpios property

`DT_SPI_DEV_HAS_CS_GPIOS(spi_dev)`

Does a SPI device have a chip select line configured? Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
```

(continues on next page)

(continued from previous page)

```
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
        };
};

spi2: spi@... {
        compatible = "vnd,spi";
        c: spi-dev-c@0 {
                reg = <0>;
        };
};
```

Example usage:

```
DT_SPI_DEV_HAS_CS_GPIOS(DT_NODELABEL(a)) // 1
DT_SPI_DEV_HAS_CS_GPIOS(DT_NODELABEL(b)) // 1
DT_SPI_DEV_HAS_CS_GPIOS(DT_NODELABEL(c)) // 0
```

Parameters

- `spi_dev` – a SPI device node identifier

Returns 1 if spi_dev's bus node *DT_BUS(spi_dev)* has a chip select pin at index *DT_REG_ADDR(spi_dev)*, 0 otherwise

`DT_SPI_DEV_CS_GPIOS_CTLR(spi_dev)`

Get a SPI device's chip select GPIO controller's node identifier.

Example devicetree fragment:

```
gpio1: gpio@... { ... };

gpio2: gpio@... { ... };

spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
        };
};
```

Example usage:

```
DT_SPI_DEV_CS_GPIOS_CTLR(DT_NODELABEL(a)) // DT_NODELABEL(gpio1)
DT_SPI_DEV_CS_GPIOS_CTLR(DT_NODELABEL(b)) // DT_NODELABEL(gpio2)
```

Parameters

- `spi_dev` – a SPI device node identifier

Returns node identifier for spi_dev's chip select GPIO controller

DT_SPI_DEV_CS_GPIOS_LABEL(spi_dev)

Get a SPI device's chip select GPIO controller's label property.

Example devicetree fragment:

```
gpio1: gpio@... {
        label = "GPIO_1";
};

gpio2: gpio@... {
        label = "GPIO_2";
};

spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
        };
};
```

Example usage:

```
DT_SPI_DEV_CS_GPIOS_LABEL(DT_NODELABEL(a)) // "GPIO_1"
DT_SPI_DEV_CS_GPIOS_LABEL(DT_NODELABEL(b)) // "GPIO_2"
```

Parameters

- `spi_dev` – a SPI device node identifier

Returns label property of spi_dev's chip select GPIO controller

DT_SPI_DEV_CS_GPIOS_PIN(spi_dev)

Get a SPI device's chip select GPIO pin number.

It's an error if the GPIO specifier for spi_dev's entry in its bus node's cs-gpios property has no pin cell.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>,
                   <&gpio2 20 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };

        b: spi-dev-b@1 {
                reg = <1>;
```

(continues on next page)

(continued from previous page)

```
        };
};
```

Example usage:

```
DT_SPI_DEV_CS_GPIOS_PIN(DT_NODELABEL(a)) // 10
DT_SPI_DEV_CS_GPIOS_PIN(DT_NODELABEL(b)) // 20
```

Parameters

- `spi_dev` – a SPI device node identifier

Returns pin number of spi_dev's chip select GPIO

`DT_SPI_DEV_CS_GPIOS_FLAGS(spi_dev)`

Get a SPI device's chip select GPIO flags.

Example devicetree fragment:

```
spi1: spi@... {
        compatible = "vnd,spi";
        cs-gpios = <&gpio1 10 GPIO_ACTIVE_LOW>;

        a: spi-dev-a@0 {
                reg = <0>;
        };
};
```

Example usage:

```
DT_SPI_DEV_CS_GPIOS_FLAGS(DT_NODELABEL(a)) // GPIO_ACTIVE_LOW
```

If the GPIO specifier for spi_dev's entry in its bus node's cs-gpios property has no flags cell, this expands to zero.

Parameters

- `spi_dev` – a SPI device node identifier

Returns flags value of spi_dev's chip select GPIO specifier, or zero if there is none

`DT_INST_SPI_DEV_HAS_CS_GPIOS(inst)`

Equivalent to *DT_SPI_DEV_HAS_CS_GPIOS(DT_DRV_INST(inst))*.

See also:

DT_SPI_DEV_HAS_CS_GPIOS()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns 1 if the instance's bus has a CS pin at index *DT_INST_REG_ADDR(inst)*, 0 otherwise

`DT_INST_SPI_DEV_CS_GPIOS_CTLR(inst)`

Get GPIO controller node identifier for a SPI device instance This is equivalent to *DT_SPI_DEV_CS_GPIOS_CTLR(DT_DRV_INST(inst))*.

See also:

DT_SPI_DEV_CS_GPIOS_CTLR()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns node identifier for instance's chip select GPIO controller

`DT_INST_SPI_DEV_CS_GPIOS_LABEL(inst)`

Get GPIO controller name for a SPI device instance This is equivalent to *DT_SPI_DEV_CS_GPIOS_LABEL(DT_DRV_INST(inst))*.

See also:

DT_SPI_DEV_CS_GPIOS_LABEL()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns label property of the instance's chip select GPIO controller

`DT_INST_SPI_DEV_CS_GPIOS_PIN(inst)`

Equivalent to *DT_SPI_DEV_CS_GPIOS_PIN(DT_DRV_INST(inst))*.

See also:

DT_SPI_DEV_CS_GPIOS_PIN()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns pin number of the instance's chip select GPIO

`DT_INST_SPI_DEV_CS_GPIOS_FLAGS(inst)`

DT_SPI_DEV_CS_GPIOS_FLAGS(DT_DRV_INST(inst)).

See also:

DT_SPI_DEV_CS_GPIOS_FLAGS()

Parameters

- `inst` – DT_DRV_COMPAT instance number

Returns flags value of the instance's chip select GPIO specifier, or zero if there is none

Chosen nodes

The special `/chosen` node contains properties whose values describe system-wide settings. The *DT_CHOSEN()* macro can be used to get a node identifier for a chosen node.

group `devicetree-generic-chosen`

Defines

`DT_CHOSEN(prop)`

Get a node identifier for a /chosen node property.

This is only valid to call if *DT_HAS_CHOSEN(prop)* is 1.

Parameters

- `prop` – lowercase-and-underscores property name for the /chosen node

Returns a node identifier for the chosen node property

`DT_HAS_CHOSEN(prop)`

Test if the devicetree has a /chosen node.

Parameters

- `prop` – lowercase-and-underscores devicetree property

Returns 1 if the chosen property exists and refers to a node, 0 otherwise

There are also conveniences for commonly used zephyr-specific properties of the `/chosen` node.

group `devicetree-zephyr`

Defines

`DT_CHOSEN_ZEPHYR_ENTROPY_LABEL`

If there is a chosen node zephyr,entropy property which has a label property, that property's value. Undefined otherwise.

`DT_CHOSEN_ZEPHYR_FLASH_CONTROLLER_LABEL`

If there is a chosen node zephyr,flash-controller property which has a label property, that property's value. Undefined otherwise.

`DT_CHOSEN_ZEPHYR_CAN_PRIMARY_LABEL`

If there is a chosen node zephyr,can-primary property which has a label property, that property's value. Undefined otherwise.

The following table documents some commonly used Zephyr-specific chosen nodes.

Sometimes, a chosen node's label property will be used to set the default value of a Kconfig option which in turn configures a hardware-specific device. This is usually for backwards compatibility in cases when the Kconfig option predates devicetree support in Zephyr. In other cases, there is no Kconfig option, and the devicetree node is used directly in the source code to select a device.

Table 4: Zephyr-specific chosen properties

Property	Purpose
zephyr,bt-c2h-uart	Selects the UART used for host communication in the bluetooth-hci-uart-sample
zephyr,bt-mon-uart	Sets UART device used for the Bluetooth monitor logging
zephyr,bt-uart	Sets UART device used by Bluetooth
zephyr,can-primary	Sets the primary CAN controller
zephyr,ccm	Core-Coupled Memory node on some STM32 SoCs
zephyr,code-partition	Flash partition that the Zephyr image's text section should be linked into
zephyr,console	Sets UART device used by console driver
zephyr,dtcm	Data Tightly Coupled Memory node on some Arm SoCs
zephyr,entropy	A device which can be used as a system-wide entropy source
zephyr,flash	A node whose `reg` is sometimes used to set the defaults for `CONFIG_FLASH_BASE_ADDRESS` and `CONFIG_FLASH_SIZE`
zephyr,flash-controller	The node corresponding to the flash controller device for the `zephyr,flash` node
zephyr,ipc	Used by the OpenAMP subsystem to specify the inter-process communication (IPC) device
zephyr,ipc_shm	A node whose `reg` is used by the OpenAMP subsystem to determine the base address and size of the shared memory (SHM) usable for interprocess-communication (IPC)
zephyr,itcm	Instruction Tightly Coupled Memory node on some Arm SoCs
zephyr,ot-uart	Used by the OpenThread to specify UART device for Spinel protocol
zephyr,shell-uart	Sets UART device used by serial shell backend
zephyr,sram	A node whose `reg` sets the base address and size of SRAM memory available to the Zephyr image, used during linking
zephyr,uart-mcumgr	UART used for *Device Management*
zephyr,uart-pipe	Sets default `CONFIG_UART_PIPE_ON_DEV_NAME`
zephyr,usb-device	USB device node. If defined and has a `vbus-gpios` property, these will be used by the USB subsystem to enable/disable VBUS

7.6.2 Bindings index

This page documents the available devicetree bindings. See *Devicetree bindings* for an introduction to the Zephyr bindings file format.

Vendor index

This section contains an index of hardware vendors. Click on a vendor's name to go to the list of bindings for that vendor.

- *Generic or vendor-independent*
- *Altera Corp. (altr)*
- *AMS AG (ams)*
- *Analog Devices, Inc. (adi)*
- *Andes Technology Corporation (andestech)*

- *Apa Electronic Co., Ltd (apa)*
- *Aptina Imaging (aptina)*
- *Arduino (arduino)*
- *ARM Ltd. (arm)*
- *Asahi Kasei Corp. (asahi-kasei)*
- *ASMedia Technology Inc. (asmedia)*
- *Atmel Corporation (atmel)*
- *Avago Technologies (avago)*
- *Bosch Sensortec GmbH (bosch)*
- *Broadcom Corporation (brcm)*
- *Cadence Design Systems Inc. (cdns)*
- *Cypress Semiconductor Corporation (cypress)*
- *Dalian Good Display Co., Ltd. (gooddisplay)*
- *Espressif Systems (espressif)*
- *Fairchild Semiconductor (fcs)*
- *FocalTech Systems Co.,Ltd (focaltech)*
- *Freescale Semiconductor (fsl)*
- *Future Technology Devices International Ltd. (ftdi)*
- *Gaisler (gaisler)*
- *GreeLed Electronic Ltd. (greeled)*
- *Guangzhou Aosong Electronic Co., Ltd. (aosong)*
- *Holtek Semiconductor, Inc. (holtek)*
- *Honeywell (honeywell)*
- *HOPERF Microelectronics Co. Ltd (hoperf)*
- *ILI Technology Corporation (ILITEK) (ilitek)*
- *Infineon Technologies (infineon)*
- *Intel Corporation (intel)*
- *Intersil (isil)*
- *InvenSense Inc. (invensense)*
- *Inventek Systems (inventek)*
- *ITE Tech. Inc. (ite)*
- *JEDEC Solid State Technology Association (jedec)*
- *Linaro Limited (linaro)*
- *LiteX SoC builder (litex)*
- *Maxim Integrated Products (maxim)*
- *Measurement Specialties (meas)*
- *Micro:bit Educational Foundation (microbit)*
- *Microchip Technology Inc. (microchip)*
- *Microchip Technology Inc. (formerly Microsemi Corporation) (microsemi)*

- *Nordic Semiconductor (nordic)*
- *Nuvoton Technology Corporation (nuvoton)*
- *NXP Semiconductors (nxp)*
- *OmniVision Technologies (ovti)*
- *open-isa.org (openisa)*
- *OpenCores.org (opencores)*
- *Panasonic Corporation (panasonic)*
- *Plantower Co., Ltd (plantower)*
- *QEMU, a generic and open source machine emulator and virtualizer (qemu)*
- *Qorvo, Inc (formerly Decawave) (decawave)*
- *Quectel Wireless Solutions Co., Ltd. (quectel)*
- *QuickLogic Corp. (quicklogic)*
- *Renesas Electronics Corporation (renesas)*
- *RISC-V Foundation (riscv)*
- *ROCKTECH DISPLAYS LIMITED (rocktech)*
- *Seeed Technology Co., Ltd (seeed)*
- *SEGGER Microcontroller GmbH (segger)*
- *Semtech Corporation (semtech)*
- *Sensirion AG (sensirion)*
- *Sharp Corporation (sharp)*
- *Sierra Wireless (swir)*
- *SiFive, Inc. (sifive)*
- *Silicon Laboratories (silabs)*
- *Sitronix Technology Corporation (sitronix)*
- *Skyworks Solutions, Inc. (skyworks)*
- *Smart Battery System (sbs)*
- *Solomon Systech Limited (solomon)*
- *Standard Microsystems Corporation (smsc)*
- *STMicroelectronics (st)*
- *Synopsys, Inc. (snps)*
- *Synopsys, Inc. (formerly ARC International PLC) (arc)*
- *Telink Semiconductor (telink)*
- *Texas Instruments (ti)*
- *u-blox (u-blox)*
- *Vishay Intertechnology, Inc (vishay)*
- *Wistron NeWeb Corporation (wnc)*
- *WIZnet Co., Ltd. (wiznet)*
- *Worldsemi Co., Limited (worldsemi)*
- *Würth Elektronik GmbH. (we)*

- *Xilinx (xlnx)*
- *Zephyr-specific binding (zephyr)*
- *Unknown vendor*

Bindings by vendor

This section contains available bindings, grouped by vendor. Within each group, bindings are listed by the "compatible" property they apply to, like this:

Vendor name (vendor prefix)

- <compatible-A>
- <compatible-B> (on <bus-name> bus)
- <compatible-C>
- ...

The text "(on <bus-name> bus)" appears when bindings may behave differently depending on the bus the node appears on. For example, this applies to some sensor device nodes, which may appear as children of either I2C or SPI bus nodes.

Generic or vendor-independent

- dtbinding_adafruit_feather_header
- dtbinding_arduino_header_r3
- dtbinding_atmel_xplained_header
- dtbinding_atmel_xplained_pro_header
- dtbinding_ethernet_phy
- dtbinding_fixed_clock
- dtbinding_fixed_rate_clock
- dtbinding_partition
- dtbinding_gpio_i2c
- dtbinding_gpio_keys
- dtbinding_gpio_leds
- dtbinding_lm75
- dtbinding_lm77
- dtbinding_mikro_bus
- dtbinding_mmio_sram
- dtbinding_ns16550
- dtbinding_particle_gen3_header
- dtbinding_pwm_leds
- dtbinding_regulator_fixed
- dtbinding_reserved_memory
- dtbinding_riscv_it8xxx2
- dtbinding_sample_controller
- dtbinding_shared_irq

- dtbinding_soc_nv_flash
- dtbinding_syscon
- dtbinding_usb_audio
- dtbinding_usb_audio_hp
- dtbinding_usb_audio_hs
- dtbinding_usb_audio_mic
- dtbinding_usb_nop_xceiv
- dtbinding_vexriscv_intc0
- dtbinding_voltage_divider

Altera Corp. (altr)

- dtbinding_altr_jtag_uart
- dtbinding_altr_msgdma
- dtbinding_altr_nios2_i2c
- dtbinding_altr_nios2f

AMS AG (ams)

- dtbinding_ams_ccs811
- dtbinding_ams_ens210
- dtbinding_ams_iaqcore

Analog Devices, Inc. (adi)

- dtbinding_adi_adt7420
- dtbinding_adi_adxl345
- dtbinding_adi_adxl362
- dtbinding_adi_adxl372_i2c
- dtbinding_adi_adxl372_spi

Andes Technology Corporation (andestech)

- dtbinding_andestech_atcgpio100

Apa Electronic Co., Ltd (apa)

- dtbinding_apa_apa_102

Aptina Imaging (aptina)

- dtbinding_aptina_mt9m114

Arduino (arduino)

- dtbinding_arduino_uno_adc

ARM Ltd. (arm)

- dtbinding_arm_armv6m_mpu
- dtbinding_arm_armv7m_mpu
- dtbinding_arm_armv8_timer
- dtbinding_arm_armv8.1m_mpu
- dtbinding_arm_armv8m_mpu
- dtbinding_arm_cmsdk_dtimer
- dtbinding_arm_cmsdk_gpio
- dtbinding_arm_cmsdk_timer
- dtbinding_arm_cmsdk_uart
- dtbinding_arm_cmsdk_watchdog
- dtbinding_arm_cortex_a53
- dtbinding_arm_cortex_a72
- dtbinding_arm_cortex_m0
- dtbinding_arm_cortex_m0+
- dtbinding_arm_cortex_m1
- dtbinding_arm_cortex_m23
- dtbinding_arm_cortex_m3
- dtbinding_arm_cortex_m33
- dtbinding_arm_cortex_m33f
- dtbinding_arm_cortex_m4
- dtbinding_arm_cortex_m4f
- dtbinding_arm_cortex_m7
- dtbinding_arm_cortex_r4
- dtbinding_arm_cortex_r4f
- dtbinding_arm_cortex_r5
- dtbinding_arm_cortex_r5f
- dtbinding_arm_cortex_r7
- dtbinding_arm_cortex_r82
- dtbinding_arm_cryptocell_310
- dtbinding_arm_cryptocell_312
- dtbinding_arm_dma_pl330
- dtbinding_arm_dtcm
- dtbinding_arm_gic
- dtbinding_arm_itcm
- dtbinding_arm_mhu
- dtbinding_arm_mps2_fpgaio_gpio
- dtbinding_arm_mps3_fpgaio_gpio
- dtbinding_arm_pl011

- dtbinding_arm_psci
- dtbinding_arm_sbsa_uart
- dtbinding_arm_scc
- dtbinding_arm_v6m_nvic
- dtbinding_arm_v7m_nvic
- dtbinding_arm_v8.1m_nvic
- dtbinding_arm_v8m_nvic
- dtbinding_arm_versatile_i2c

Asahi Kasei Corp. (asahi-kasei)

- dtbinding_asahi_kasei_ak8975

ASMedia Technology Inc. (asmedia)

- dtbinding_asmedia_asmedia2364

Atmel Corporation (atmel)

- dtbinding_atmel_at24
- dtbinding_atmel_at25
- dtbinding_atmel_at45
- dtbinding_atmel_rf2xx
- dtbinding_atmel_sam_afec
- dtbinding_atmel_sam_dac
- dtbinding_atmel_sam_flash_controller
- dtbinding_atmel_sam_gmac
- dtbinding_atmel_sam_gpio
- dtbinding_atmel_sam_i2c_twi
- dtbinding_atmel_sam_i2c_twihs
- dtbinding_atmel_sam_i2c_twim
- dtbinding_atmel_sam_mdio
- dtbinding_atmel_sam_pinctrl
- dtbinding_atmel_sam_pwm
- dtbinding_atmel_sam_spi
- dtbinding_atmel_sam_ssc
- dtbinding_atmel_sam_tc
- dtbinding_atmel_sam_tc_qdec
- dtbinding_atmel_sam_trng
- dtbinding_atmel_sam_uart
- dtbinding_atmel_sam_usart
- dtbinding_atmel_sam_usbc

- dtbinding_atmel_sam_usbhs
- dtbinding_atmel_sam_watchdog
- dtbinding_atmel_sam_xdmac
- dtbinding_atmel_sam0_adc
- dtbinding_atmel_sam0_dac
- dtbinding_atmel_sam0_dmac
- dtbinding_atmel_sam0_eic
- dtbinding_atmel_sam0_gmac
- dtbinding_atmel_sam0_gpio
- dtbinding_atmel_sam0_i2c
- dtbinding_atmel_sam0_device_id
- dtbinding_atmel_sam0_nvmctrl
- dtbinding_atmel_sam0_pinctrl
- dtbinding_atmel_sam0_pinmux
- dtbinding_atmel_sam0_rtc
- dtbinding_atmel_sam0_sercom
- dtbinding_atmel_sam0_spi
- dtbinding_atmel_sam0_tc32
- dtbinding_atmel_sam0_tcc_pwm
- dtbinding_atmel_sam0_uart
- dtbinding_atmel_sam0_usb
- dtbinding_atmel_sam0_watchdog
- dtbinding_atmel_sam4l_gpio
- dtbinding_atmel_sam4l_uid
- dtbinding_atmel_samd2x_gclk
- dtbinding_atmel_samd2x_pm
- dtbinding_atmel_samd5x_gclk
- dtbinding_atmel_samd5x_mclk
- dtbinding_atmel_winc1500

Avago Technologies (avago)

- dtbinding_avago_apds9960

Bosch Sensortec GmbH (bosch)

- dtbinding_bosch_bma280
- dtbinding_bosch_bmc150_magn
- dtbinding_bosch_bme280_spi
- dtbinding_bosch_bme280_i2c
- dtbinding_bosch_bme680_i2c

- dtbinding_bosch_bmg160
- dtbinding_bosch_bmi160
- dtbinding_bosch_bmi270_i2c
- dtbinding_bosch_bmm150
- dtbinding_bosch_bmp388_i2c
- dtbinding_bosch_bmp388_spi
- dtbinding_bosch_mcan
- dtbinding_bosch_m_can_base

Broadcom Corporation (brcm)

- dtbinding_brcm_iproc_pax_dma_v1
- dtbinding_brcm_iproc_pax_dma_v2
- dtbinding_brcm_iproc_pcie_ep

Cadence Design Systems Inc. (cdns)

- dtbinding_cadence_tensilica_xtensa_lx4
- dtbinding_cadence_tensilica_xtensa_lx6
- dtbinding_cdns_xtensa_core_intc

Cypress Semiconductor Corporation (cypress)

- dtbinding_cypress_cy8c95xx_gpio
- dtbinding_cypress_cy8c95xx_gpio_port
- dtbinding_cypress_psoc6_flash_controller
- dtbinding_cypress_psoc6_gpio
- dtbinding_cypress_psoc6_hsiom
- dtbinding_cypress_psoc6_int_mux
- dtbinding_cypress_psoc6_intmux_ch
- dtbinding_cypress_psoc6_pinctrl
- dtbinding_cypress_psoc6_spi
- dtbinding_cypress_psoc6_uart
- dtbinding_cypress_psoc6_uid

Dalian Good Display Co., Ltd. (gooddisplay)

- dtbinding_gooddisplay_gd7965

Espressif Systems (espressif)

- dtbinding_espressif_esp_at
- dtbinding_espressif_esp32_flash_controller
- dtbinding_espressif_esp32_gpio
- dtbinding_espressif_esp32_i2c
- dtbinding_espressif_esp32_intc
- dtbinding_espressif_esp32_pinmux
- dtbinding_espressif_esp32_rtc
- dtbinding_espressif_esp32_spi
- dtbinding_espressif_esp32_trng
- dtbinding_espressif_esp32_uart
- dtbinding_espressif_esp32_watchdog
- dtbinding_espressif_esp32c3_uart
- dtbinding_espressif_esp32s2_uart

Fairchild Semiconductor (fcs)

- dtbinding_fcs_fxl6408

FocalTech Systems Co.,Ltd (focaltech)

- dtbinding_focaltech_ft5336

Freescale Semiconductor (fsl)

- dtbinding_fsl_imx21_i2c
- dtbinding_fsl_imx6sx_lcdif
- dtbinding_fsl_imx7d_pwm

Future Technology Devices International Ltd. (ftdi)

- dtbinding_ftdi_ft800

Gaisler (gaisler)

- dtbinding_gaisler_apbuart
- dtbinding_gaisler_gptimer
- dtbinding_gaisler_irqmp

GreeLed Electronic Ltd. (greeled)

- dtbinding_greeled_lpd8803
- dtbinding_greeled_lpd8806

Guangzhou Aosong Electronic Co., Ltd. (aosong)

- dtbinding_aosong_dht

Holtek Semiconductor, Inc. (holtek)

- dtbinding_holtek_ht16k33
- dtbinding_holtek_ht16k33_keyscan

Honeywell (honeywell)

- dtbinding_honeywell_hmc5883l
- dtbinding_honeywell_mpr
- dtbinding_honeywell_sm351lt

HOPERF Microelectronics Co. Ltd (hoperf)

- dtbinding_hoperf_hp206c
- dtbinding_hoperf_th02

ILI Technology Corporation (ILITEK) (ilitek)

- dtbinding_ilitek_ili9340
- dtbinding_ilitek_ili9341
- dtbinding_ilitek_ili9488

Infineon Technologies (infineon)

- dtbinding_infineon_dps310
- dtbinding_infineon_xmc4xxx_uart

Intel Corporation (intel)

- dtbinding_intel_adsp_mailbox
- dtbinding_intel_apollo_lake
- dtbinding_intel_atom
- dtbinding_intel_cavs_i2s
- dtbinding_intel_cavs_idc
- dtbinding_intel_cavs_intc
- dtbinding_intel_dmic
- dtbinding_intel_e1000
- dtbinding_intel_elkhart_lake
- dtbinding_intel_gna
- dtbinding_intel_gpio
- dtbinding_intel_hpet
- dtbinding_intel_ibecc
- dtbinding_intel_ioapic
- dtbinding_intel_pcie
- dtbinding_intel_s1000_pinmux
- dtbinding_intel_vt_d

- dtbinding_intel_x86

Intersil (isil)

- dtbinding_isil_isl29035

InvenSense Inc. (invensense)

- dtbinding_invensense_icm42605
- dtbinding_invensense_mpu6050
- dtbinding_invensense_mpu9150

Inventek Systems (inventek)

- dtbinding_inventek_eswifi
- dtbinding_inventek_eswifi_uart

ITE Tech. Inc. (ite)

- dtbinding_ite_it8xxx2_adc
- dtbinding_ite_it8xxx2_bbram
- dtbinding_ite_it8xxx2_flash_controller
- dtbinding_ite_it8xxx2_gpio
- dtbinding_ite_it8xxx2_i2c
- dtbinding_ite_it8xxx2_intc
- dtbinding_ite_it8xxx2_pinctrl_conf
- dtbinding_ite_it8xxx2_pinmux
- dtbinding_ite_it8xxx2_pwm
- dtbinding_ite_it8xxx2_pwmprs
- dtbinding_ite_it8xxx2_sspi
- dtbinding_ite_it8xxx2_timer
- dtbinding_ite_it8xxx2_watchdog

JEDEC Solid State Technology Association (jedec)

- dtbinding_jedec_spi_nor

Linaro Limited (linaro)

- dtbinding_96boards_lscon_1v8
- dtbinding_96boards_lscon_3v3

LiteX SoC builder (litex)

- dtbinding_litex_clk
- dtbinding_litex_clkout
- dtbinding_litex_dna0
- dtbinding_litex_eth0
- dtbinding_litex_gpio
- dtbinding_litex_i2c
- dtbinding_litex_i2s
- dtbinding_litex_prbs
- dtbinding_litex_pwm
- dtbinding_litex_spi
- dtbinding_litex_timer0
- dtbinding_litex_uart0

Maxim Integrated Products (maxim)

- dtbinding_maxim_ds3231
- dtbinding_maxim_max17055
- dtbinding_maxim_max17262
- dtbinding_maxim_max30101
- dtbinding_maxim_max44009
- dtbinding_maxim_max6675

Measurement Specialties (meas)

- dtbinding_meas_ms5607_i2c
- dtbinding_meas_ms5607_spi
- dtbinding_meas_ms5837

Micro:bit Educational Foundation (microbit)

- dtbinding_microbit_edge_connector

Microchip Technology Inc. (microchip)

- dtbinding_microchip_enc28j60
- dtbinding_microchip_enc424j600
- dtbinding_microchip_ksz8794
- dtbinding_microchip_ksz8863
- dtbinding_microchip_mcp23s17
- dtbinding_microchip_mcp2515
- dtbinding_microchip_mcp3204
- dtbinding_microchip_mcp3208

- dtbinding_microchip_mcp4725
- dtbinding_microchip_mcp7940n
- dtbinding_microchip_mcp9808
- dtbinding_microchip_xec_adc
- dtbinding_microchip_xec_adc_v2
- dtbinding_microchip_xec_ecia
- dtbinding_microchip_xec_ecia_girq
- dtbinding_microchip_xec_espi
- dtbinding_microchip_xec_espi_saf
- dtbinding_microchip_xec_gpio
- dtbinding_microchip_xec_gpio_v2
- dtbinding_microchip_xec_i2c
- dtbinding_microchip_xec_i2c_v2
- dtbinding_microchip_xec_kscan
- dtbinding_microchip_xec_pcr
- dtbinding_microchip_xec_peci
- dtbinding_microchip_xec_pinmux
- dtbinding_microchip_xec_ps2
- dtbinding_microchip_xec_pwm
- dtbinding_microchip_xec_qmspi
- dtbinding_microchip_xec_rtos_timer
- dtbinding_microchip_xec_tach
- dtbinding_microchip_xec_timer
- dtbinding_microchip_xec_uart
- dtbinding_microchip_xec_watchdog

Microchip Technology Inc. (formerly Microsemi Corporation) (microsemi)

- dtbinding_microsemi_coreuart

Nordic Semiconductor (nordic)

- dtbinding_nordic_nrf_adc
- dtbinding_nordic_nrf_cc310
- dtbinding_nordic_nrf_cc312
- dtbinding_nordic_nrf_clock
- dtbinding_nordic_nrf_dppic
- dtbinding_nordic_nrf_ecb
- dtbinding_nordic_nrf_egu
- dtbinding_nordic_nrf_ficr
- dtbinding_nordic_nrf_gpio

- dtbinding_nordic_nrf_gpiote
- dtbinding_nordic_nrf_i2s
- dtbinding_nordic_nrf_ipc
- dtbinding_nordic_nrf_kmu
- dtbinding_nordic_nrf_pdm
- dtbinding_nordic_nrf_power
- dtbinding_nordic_nrf_pwm
- dtbinding_nordic_nrf_qdec
- dtbinding_nordic_nrf_qspi
- dtbinding_nordic_nrf_radio
- dtbinding_nordic_nrf_regulators
- dtbinding_nordic_nrf_rng
- dtbinding_nordic_nrf_rtc
- dtbinding_nordic_nrf_saadc
- dtbinding_nordic_nrf_spi
- dtbinding_nordic_nrf_spim
- dtbinding_nordic_nrf_spis
- dtbinding_nordic_nrf_spu
- dtbinding_nordic_nrf_sw_pwm
- dtbinding_nordic_nrf_temp
- dtbinding_nordic_nrf_timer
- dtbinding_nordic_nrf_twi
- dtbinding_nordic_nrf_twim
- dtbinding_nordic_nrf_twis
- dtbinding_nordic_nrf_uart
- dtbinding_nordic_nrf_uarte
- dtbinding_nordic_nrf_uicr
- dtbinding_nordic_nrf_usbd
- dtbinding_nordic_nrf_vmc
- dtbinding_nordic_nrf_watchdog
- dtbinding_nordic_nrf21540_fem
- dtbinding_nordic_nrf21540_fem_spi
- dtbinding_nordic_nrf51_flash_controller
- dtbinding_nordic_nrf52_flash_controller
- dtbinding_nordic_nrf53_flash_controller
- dtbinding_nordic_nrf91_flash_controller
- dtbinding_nordic_qspi_nor

Nuvoton Technology Corporation (nuvoton)

- dtbinding_nuvoton_npcx_adc
- dtbinding_nuvoton_npcx_bbram
- dtbinding_nuvoton_npcx_booter_variant
- dtbinding_nuvoton_npcx_espi
- dtbinding_nuvoton_npcx_espi_vw_conf
- dtbinding_nuvoton_npcx_gpio
- dtbinding_nuvoton_npcx_host_sub
- dtbinding_nuvoton_npcx_host_uart
- dtbinding_nuvoton_npcx_i2c_ctrl
- dtbinding_nuvoton_npcx_i2c_port
- dtbinding_nuvoton_npcx_itim_timer
- dtbinding_nuvoton_npcx_lvolctrl_conf
- dtbinding_nuvoton_npcx_lvolctrl_def
- dtbinding_nuvoton_npcx_miwu
- dtbinding_nuvoton_npcx_miwu_int_map
- dtbinding_nuvoton_npcx_miwu_wui_map
- dtbinding_nuvoton_npcx_pcc
- dtbinding_nuvoton_npcx_pinctrl_conf
- dtbinding_nuvoton_npcx_pinctrl_def
- dtbinding_nuvoton_npcx_ps2_channel
- dtbinding_nuvoton_npcx_ps2_ctrl
- dtbinding_nuvoton_npcx_psl_out
- dtbinding_nuvoton_npcx_pslctrl_conf
- dtbinding_nuvoton_npcx_pslctrl_def
- dtbinding_nuvoton_npcx_pwm
- dtbinding_nuvoton_npcx_scfg
- dtbinding_nuvoton_npcx_soc_id
- dtbinding_nuvoton_npcx_tach
- dtbinding_nuvoton_npcx_uart
- dtbinding_nuvoton_npcx_watchdog
- dtbinding_nuvoton_numicro_uart

NXP Semiconductors (nxp)

- dtbinding_nxp_flexpwm
- dtbinding_nxp_fxas21002
- dtbinding_nxp_fxos8700
- dtbinding_nxp_imx_ccm
- dtbinding_nxp_imx_ccm_rev2

- dtbinding_nxp_imx_csi
- dtbinding_nxp_imx_dtcm
- dtbinding_nxp_imx_epit
- dtbinding_nxp_imx_flexspi
- dtbinding_nxp_imx_flexspi_device
- dtbinding_nxp_imx_flexspi_hyperflash
- dtbinding_nxp_imx_flexspi_hyperram
- dtbinding_nxp_imx_flexspi_mx25um51345g
- dtbinding_nxp_imx_flexspi_nor
- dtbinding_nxp_imx_gpio
- dtbinding_nxp_imx_gpt
- dtbinding_nxp_imx_itcm
- dtbinding_nxp_imx_iuart
- dtbinding_nxp_imx_lpi2c
- dtbinding_nxp_imx_lpspi
- dtbinding_nxp_imx_mu
- dtbinding_nxp_imx_pwm
- dtbinding_nxp_imx_semc
- dtbinding_nxp_imx_uart
- dtbinding_nxp_imx_usdhc
- dtbinding_nxp_imx_wdog
- dtbinding_nxp_kinetis_acmp
- dtbinding_nxp_kinetis_adc12
- dtbinding_nxp_kinetis_adc16
- dtbinding_nxp_kinetis_dac
- dtbinding_nxp_kinetis_dac32
- dtbinding_nxp_kinetis_dspi
- dtbinding_nxp_kinetis_ethernet
- dtbinding_nxp_kinetis_flexcan
- dtbinding_nxp_kinetis_ftfa
- dtbinding_nxp_kinetis_ftfe
- dtbinding_nxp_kinetis_ftfl
- dtbinding_nxp_kinetis_ftm
- dtbinding_nxp_kinetis_ftm_pwm
- dtbinding_nxp_kinetis_gpio
- dtbinding_nxp_kinetis_i2c
- dtbinding_nxp_kinetis_ke1xf_sim
- dtbinding_nxp_kinetis_lpsci
- dtbinding_nxp_kinetis_lptmr

- dtbinding_nxp_kinetis_lpuart
- dtbinding_nxp_kinetis_mcg
- dtbinding_nxp_kinetis_pcc
- dtbinding_nxp_kinetis_pinmux
- dtbinding_nxp_kinetis_pit
- dtbinding_nxp_kinetis_ptp
- dtbinding_nxp_kinetis_pwt
- dtbinding_nxp_kinetis_rnga
- dtbinding_nxp_kinetis_rtc
- dtbinding_nxp_kinetis_scg
- dtbinding_nxp_kinetis_sim
- dtbinding_nxp_kinetis_temperature
- dtbinding_nxp_kinetis_tpm
- dtbinding_nxp_kinetis_trng
- dtbinding_nxp_kinetis_uart
- dtbinding_nxp_kinetis_usbd
- dtbinding_nxp_kinetis_wdog
- dtbinding_nxp_kinetis_wdog32
- dtbinding_nxp_lpc_ctimer
- dtbinding_nxp_lpc_dma
- dtbinding_nxp_lpc_flexcomm
- dtbinding_nxp_lpc_gpio
- dtbinding_nxp_lpc_i2c
- dtbinding_nxp_lpc_i2s
- dtbinding_nxp_lpc_iap
- dtbinding_nxp_lpc_iocon
- dtbinding_nxp_lpc_iocon_pio
- dtbinding_nxp_lpc_lpadc
- dtbinding_nxp_lpc_mailbox
- dtbinding_nxp_lpc_rng
- dtbinding_nxp_lpc_spi
- dtbinding_nxp_lpc_syscon
- dtbinding_nxp_lpc_uid
- dtbinding_nxp_lpc_usart
- dtbinding_nxp_lpc_wwdt
- dtbinding_nxp_lpc11u6x_eeprom
- dtbinding_nxp_lpc11u6x_gpio
- dtbinding_nxp_lpc11u6x_i2c
- dtbinding_nxp_lpc11u6x_pinmux

- dtbinding_nxp_lpc11u6x_syscon
- dtbinding_nxp_lpc11u6x_uart
- dtbinding_nxp_mcr20a
- dtbinding_nxp_mcux_edma
- dtbinding_nxp_mcux_usbd
- dtbinding_nxp_pca95xx
- dtbinding_nxp_pca9633
- dtbinding_nxp_pcal6408a
- dtbinding_nxp_sctimer_pwm

OmniVision Technologies (ovti)

- dtbinding_ovti_ov2640
- dtbinding_ovti_ov7725

open-isa.org (openisa)

- dtbinding_openisa_rv32m1_event_unit
- dtbinding_openisa_rv32m1_ftfe
- dtbinding_openisa_rv32m1_genfsk
- dtbinding_openisa_rv32m1_gpio
- dtbinding_openisa_rv32m1_intmux
- dtbinding_openisa_rv32m1_intmux_ch
- dtbinding_openisa_rv32m1_lpi2c
- dtbinding_openisa_rv32m1_lpspi
- dtbinding_openisa_rv32m1_lptmr
- dtbinding_openisa_rv32m1_lpuart
- dtbinding_openisa_rv32m1_pcc
- dtbinding_openisa_rv32m1_pinmux
- dtbinding_openisa_rv32m1_tpm
- dtbinding_openisa_rv32m1_trng

OpenCores.org (opencores)

- dtbinding_opencores_spi_simple

Panasonic Corporation (panasonic)

- dtbinding_panasonic_amg88xx

Plantower Co., Ltd (plantower)

- dtbinding_plantower_pms7003

QEMU, a generic and open source machine emulator and virtualizer (qemu)

- dtbinding_qemu_ivshmem
- dtbinding_qemu_nios2_zephyr

Qorvo, Inc (formerly Decawave) (decawave)

- dtbinding_decawave_dw1000

Quectel Wireless Solutions Co., Ltd. (quectel)

- dtbinding_quectel_bg9x

QuickLogic Corp. (quicklogic)

- dtbinding_quicklogic_eos_s3_gpio

Renesas Electronics Corporation (renesas)

- dtbinding_renesas_rcar_can
- dtbinding_renesas_rcar_cmt
- dtbinding_renesas_rcar_cpg_mssr
- dtbinding_renesas_rcar_gpio
- dtbinding_renesas_rcar_i2c
- dtbinding_renesas_rcar_scif

RISC-V Foundation (riscv)

- dtbinding_riscv_clint0
- dtbinding_riscv_cpu_intc
- dtbinding_riscv_sifive_e24

ROCKTECH DISPLAYS LIMITED (rocktech)

- dtbinding_rocktech_rk043fn02h_ct

Seeed Technology Co., Ltd (seeed)

- dtbinding_seeed_grove_light
- dtbinding_seeed_grove_temperature

SEGGER Microcontroller GmbH (segger)

- dtbinding_segger_rtt_uart

Semtech Corporation (semtech)

- dtbinding_semtech_sx1261
- dtbinding_semtech_sx1262
- dtbinding_semtech_sx1272
- dtbinding_semtech_sx1276
- dtbinding_semtech_sx1509b_gpio
- dtbinding_semtech_sx9500

Sensirion AG (sensirion)

- dtbinding_sensirion_sgp40
- dtbinding_sensirion_sht3xd
- dtbinding_sensirion_sht4x
- dtbinding_sensirion_shtcx

Sharp Corporation (sharp)

- dtbinding_sharp_ls0xx

Sierra Wireless (swir)

- dtbinding_swir_hl7800

SiFive, Inc. (sifive)

- dtbinding_sifive_dtim0
- dtbinding_sifive_gpio0
- dtbinding_sifive_i2c0
- dtbinding_sifive_iof
- dtbinding_sifive_plic_1.0.0
- dtbinding_sifive_pwm0
- dtbinding_sifive_spi0
- dtbinding_sifive_uart0
- dtbinding_sifive_wdt

Silicon Laboratories (silabs)

- dtbinding_silabs_gecko_ethernet
- dtbinding_silabs_gecko_flash_controller
- dtbinding_silabs_gecko_gpio
- dtbinding_silabs_gecko_gpio_port
- dtbinding_silabs_gecko_i2c
- dtbinding_silabs_gecko_leuart
- dtbinding_silabs_gecko_pwm
- dtbinding_silabs_gecko_rtcc

- dtbinding_silabs_gecko_spi_usart
- dtbinding_silabs_gecko_timer
- dtbinding_silabs_gecko_trng
- dtbinding_silabs_gecko_uart
- dtbinding_silabs_gecko_usart
- dtbinding_silabs_gecko_wdog
- dtbinding_silabs_si7006
- dtbinding_silabs_si7055
- dtbinding_silabs_si7060
- dtbinding_silabs_si7210

Sitronix Technology Corporation (sitronix)

- dtbinding_sitronix_st7735r
- dtbinding_sitronix_st7789v

Skyworks Solutions, Inc. (skyworks)

- dtbinding_skyworks_sky13351

Smart Battery System (sbs)

- dtbinding_sbs_sbs_gauge

Solomon Systech Limited (solomon)

- dtbinding_solomon_ssd1306fb_i2c
- dtbinding_solomon_ssd1306fb_spi
- dtbinding_solomon_ssd16xx

Standard Microsystems Corporation (smsc)

- dtbinding_smsc_lan9220

STMicroelectronics (st)

- dtbinding_st_hts221
- dtbinding_st_iis2dh_i2c
- dtbinding_st_iis2dh_spi
- dtbinding_st_iis2dlpc_spi
- dtbinding_st_iis2dlpc_i2c
- dtbinding_st_iis2iclx_spi
- dtbinding_st_iis2iclx_i2c
- dtbinding_st_iis2mdc_i2c
- dtbinding_st_iis2mdc_spi
- dtbinding_st_iis3dhhc_spi

- dtbinding_st_ism330dhcx_i2c
- dtbinding_st_ism330dhcx_spi
- dtbinding_st_lis2dh_spi
- dtbinding_st_lis2dh_i2c
- dtbinding_st_lis2dh12_i2c
- dtbinding_st_lis2ds12_spi
- dtbinding_st_lis2ds12_i2c
- dtbinding_st_lis2dw12_i2c
- dtbinding_st_lis2dw12_spi
- dtbinding_st_lis2mdl_spi
- dtbinding_st_lis2mdl_i2c
- dtbinding_st_lis3dh_i2c
- dtbinding_st_lis3mdl_magn
- dtbinding_st_lps22hb_press
- dtbinding_st_lps22hh_spi
- dtbinding_st_lps22hh_i2c
- dtbinding_st_lps25hb_press
- dtbinding_st_lsm303agr_accel_spi
- dtbinding_st_lsm303agr_accel_i2c
- dtbinding_st_lsm303dlhc_accel
- dtbinding_st_lsm303dlhc_magn
- dtbinding_st_lsm6ds0
- dtbinding_st_lsm6dsl_spi
- dtbinding_st_lsm6dsl_i2c
- dtbinding_st_lsm6dso_spi
- dtbinding_st_lsm6dso_i2c
- dtbinding_st_lsm9ds0_gyro_i2c
- dtbinding_st_lsm9ds0_mfd_i2c
- dtbinding_st_mpxxdtyy_i2s
- dtbinding_st_stm32_adc
- dtbinding_st_stm32_aes
- dtbinding_st_stm32_backup_sram
- dtbinding_st_stm32_can
- dtbinding_st_stm32_ccm
- dtbinding_st_stm32_cryp
- dtbinding_st_stm32_dac
- dtbinding_st_stm32_dma
- dtbinding_st_stm32_dma_v1
- dtbinding_st_stm32_dma_v2

- dtbinding_st_stm32_dma_v2bis
- dtbinding_st_stm32_dmamux
- dtbinding_st_stm32_eeprom
- dtbinding_st_stm32_ethernet
- dtbinding_st_stm32_exti
- dtbinding_st_stm32_fdcan
- dtbinding_st_stm32_flash_controller
- dtbinding_st_stm32_fmc
- dtbinding_st_stm32_fmc_sdram
- dtbinding_st_stm32_gpio
- dtbinding_st_stm32_hse_clock
- dtbinding_st_stm32_hsem_mailbox
- dtbinding_st_stm32_i2c_v1
- dtbinding_st_stm32_i2c_v2
- dtbinding_st_stm32_i2s
- dtbinding_st_stm32_ipcc_mailbox
- dtbinding_st_stm32_lptim
- dtbinding_st_stm32_lpuart
- dtbinding_st_stm32_msi_clock
- dtbinding_st_stm32_nv_flash
- dtbinding_st_stm32_otgfs
- dtbinding_st_stm32_otghs
- dtbinding_st_stm32_pinctrl
- dtbinding_st_stm32_pwm
- dtbinding_st_stm32_qspi
- dtbinding_st_stm32_qspi_nor
- dtbinding_st_stm32_rcc
- dtbinding_st_stm32_rng
- dtbinding_st_stm32_rtc
- dtbinding_st_stm32_sdmmc
- dtbinding_st_stm32_spi
- dtbinding_st_stm32_spi_fifo
- dtbinding_st_stm32_spi_subghz
- dtbinding_st_stm32_temp
- dtbinding_st_stm32_timers
- dtbinding_st_stm32_uart
- dtbinding_st_stm32_usart
- dtbinding_st_stm32_usb
- dtbinding_st_stm32_usbphyc

- dtbinding_st_stm32_watchdog
- dtbinding_st_stm32_window_watchdog
- dtbinding_st_stm32f0_flash_controller
- dtbinding_st_stm32f0_pll_clock
- dtbinding_st_stm32f0_rcc
- dtbinding_st_stm32f1_flash_controller
- dtbinding_st_stm32f1_pinctrl
- dtbinding_st_stm32f1_pll_clock
- dtbinding_st_stm32f100_pll_clock
- dtbinding_st_stm32f105_pll_clock
- dtbinding_st_stm32f105_pll2_clock
- dtbinding_st_stm32f2_flash_controller
- dtbinding_st_stm32f2_pll_clock
- dtbinding_st_stm32f3_flash_controller
- dtbinding_st_stm32f4_flash_controller
- dtbinding_st_stm32f4_pll_clock
- dtbinding_st_stm32f7_flash_controller
- dtbinding_st_stm32f7_pll_clock
- dtbinding_st_stm32g0_flash_controller
- dtbinding_st_stm32g0_pll_clock
- dtbinding_st_stm32g4_flash_controller
- dtbinding_st_stm32g4_pll_clock
- dtbinding_st_stm32h7_flash_controller
- dtbinding_st_stm32h7_hsi_clock
- dtbinding_st_stm32h7_pll_clock
- dtbinding_st_stm32h7_rcc
- dtbinding_st_stm32l0_flash_controller
- dtbinding_st_stm32l0_msi_clock
- dtbinding_st_stm32l0_pll_clock
- dtbinding_st_stm32l1_flash_controller
- dtbinding_st_stm32l4_flash_controller
- dtbinding_st_stm32l4_pll_clock
- dtbinding_st_stm32l5_flash_controller
- dtbinding_st_stm32u5_flash_controller
- dtbinding_st_stm32u5_msi_clock
- dtbinding_st_stm32u5_pll_clock
- dtbinding_st_stm32u5_rcc
- dtbinding_st_stm32wb_flash_controller
- dtbinding_st_stm32wb_pll_clock

- dtbinding_st_stm32wb_rcc
- dtbinding_st_stm32wl_hse_clock
- dtbinding_st_stm32wl_rcc
- dtbinding_st_stm32wl_subghz_radio
- dtbinding_st_stmpe1600
- dtbinding_st_stts751_i2c
- dtbinding_st_vl53l0x

Synopsys, Inc. (snps)

- dtbinding_snps_arcem
- dtbinding_snps_archs_idu_intc
- dtbinding_snps_arcv2_intc
- dtbinding_snps_creg_gpio
- dtbinding_snps_creg_gpio_mux_hsdk
- dtbinding_snps_designware_dma
- dtbinding_snps_designware_gpio
- dtbinding_snps_designware_i2c
- dtbinding_snps_designware_intc
- dtbinding_snps_designware_pwm
- dtbinding_snps_designware_spi
- dtbinding_snps_designware_usb
- dtbinding_snps_nsim_uart

Synopsys, Inc. (formerly ARC International PLC) (arc)

- dtbinding_arc_dccm
- dtbinding_arc_iccm

Telink Semiconductor (telink)

- dtbinding_telink_b91
- dtbinding_telink_b91_flash_controller
- dtbinding_telink_b91_gpio
- dtbinding_telink_b91_i2c
- dtbinding_telink_b91_pinmux
- dtbinding_telink_b91_power
- dtbinding_telink_b91_pwm
- dtbinding_telink_b91_spi
- dtbinding_telink_b91_trng
- dtbinding_telink_b91_uart
- dtbinding_telink_b91_zb

Texas Instruments (ti)

- dtbinding_ti_boosterpack_header
- dtbinding_ti_bq274xx
- dtbinding_ti_cc1200
- dtbinding_ti_cc13xx_cc26xx_gpio
- dtbinding_ti_cc13xx_cc26xx_i2c
- dtbinding_ti_cc13xx_cc26xx_pinmux
- dtbinding_ti_cc13xx_cc26xx_rtc
- dtbinding_ti_cc13xx_cc26xx_spi
- dtbinding_ti_cc13xx_cc26xx_trng
- dtbinding_ti_cc13xx_cc26xx_uart
- dtbinding_ti_cc2520
- dtbinding_ti_cc32xx_adc
- dtbinding_ti_cc32xx_gpio
- dtbinding_ti_cc32xx_i2c
- dtbinding_ti_cc32xx_uart
- dtbinding_ti_cc32xx_watchdog
- dtbinding_ti_dac43608
- dtbinding_ti_dac53608
- dtbinding_ti_dac60508
- dtbinding_ti_dac70508
- dtbinding_ti_dac80508
- dtbinding_ti_fdc2x1x
- dtbinding_ti_hdc
- dtbinding_ti_hdc2010
- dtbinding_ti_hdc2021
- dtbinding_ti_hdc2022
- dtbinding_ti_hdc2080
- dtbinding_ti_hdc20xx
- dtbinding_ti_ina219
- dtbinding_ti_ina23x
- dtbinding_ti_lmp90077
- dtbinding_ti_lmp90078
- dtbinding_ti_lmp90079
- dtbinding_ti_lmp90080
- dtbinding_ti_lmp90097
- dtbinding_ti_lmp90098
- dtbinding_ti_lmp90099
- dtbinding_ti_lmp90100

- dtbinding_ti_lmp90xxx_gpio
- dtbinding_ti_lp3943
- dtbinding_ti_lp503x
- dtbinding_ti_lp5562
- dtbinding_ti_msp432p4xx_uart
- dtbinding_ti_opt3001
- dtbinding_ti_stellaris_ethernet
- dtbinding_ti_stellaris_flash_controller
- dtbinding_ti_stellaris_gpio
- dtbinding_ti_stellaris_uart
- dtbinding_ti_tca9538_gpio
- dtbinding_ti_tca9546a
- dtbinding_ti_tlc59108
- dtbinding_ti_tlv320dac
- dtbinding_ti_tmp007
- dtbinding_ti_tmp112
- dtbinding_ti_tmp116

u-blox (u-blox)

- dtbinding_u_blox_sara_r4

Vishay Intertechnology, Inc (vishay)

- dtbinding_vishay_vcnl4040

Wistron NeWeb Corporation (wnc)

- dtbinding_wnc_m14a2a

WIZnet Co., Ltd. (wiznet)

- dtbinding_wiznet_w5500

Worldsemi Co., Limited (worldsemi)

- dtbinding_worldsemi_ws2812_gpio
- dtbinding_worldsemi_ws2812_spi

Würth Elektronik GmbH. (we)

- dtbinding_we_wsen_itds

Xilinx (xlnx)

- dtbinding_xlnx_gem
- dtbinding_xlnx_ttcps
- dtbinding_xlnx_xlnx_xps_gpio_1.00.a
- dtbinding_xlnx_xlnx_xps_gpio_1.00.a_gpio2
- dtbinding_xlnx_xps_spi_2.00.a
- dtbinding_xlnx_xps_timer_1.00.a
- dtbinding_xlnx_xps_timer_1.00.a_pwm
- dtbinding_xlnx_xps_uartlite_1.00.a
- dtbinding_xlnx_uartps

Zephyr-specific binding (zephyr)

- dtbinding_zephyr_adc_emul
- dtbinding_zephyr_bbram_emul
- dtbinding_zephyr_bt_hci_spi
- dtbinding_zephyr_bt_hci_spi_slave
- dtbinding_zephyr_cdc_acm_uart
- dtbinding_zephyr_emu_eeprom
- dtbinding_zephyr_espi_emul
- dtbinding_zephyr_fstab
- dtbinding_zephyr_fstab_littlefs
- dtbinding_zephyr_gpio_emul
- dtbinding_zephyr_gsm_ppp
- dtbinding_zephyr_i2c_emul
- dtbinding_zephyr_ipm_console
- dtbinding_mmc_spi_slot
- dtbinding_zephyr_modbus_serial
- dtbinding_zephyr_native_posix_rng
- dtbinding_zephyr_native_posix_uart
- dtbinding_zephyr_native_posix_udc
- dtbinding_state
- dtbinding_zephyr_sim_ec_host_cmd_periph
- dtbinding_zephyr_sim_eeprom
- dtbinding_zephyr_sim_flash
- dtbinding_zephyr_spi_emul

Unknown vendor

- dtbinding_swerv_pic

7.7 Device Driver Model

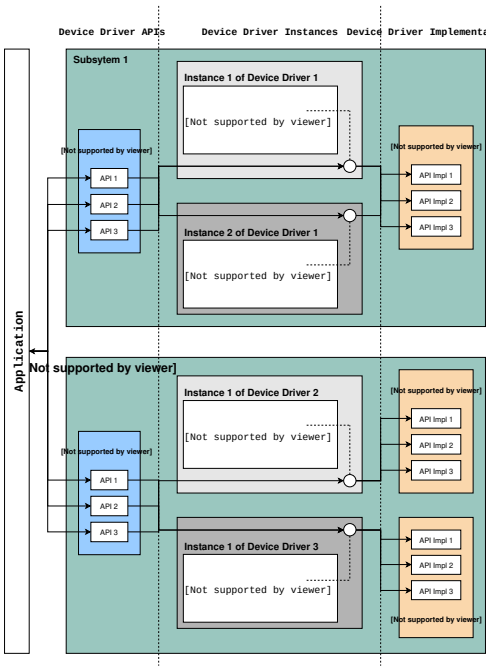
7.7.1 Introduction

The Zephyr kernel supports a variety of device drivers. Whether a driver is available depends on the board and the driver.

The Zephyr device model provides a consistent device model for configuring the drivers that are part of a system. The device model is responsible for initializing all the drivers configured into the system.

Each type of driver (e.g. UART, SPI, I2C) is supported by a generic type API.

In this model the driver fills in the pointer to the structure containing the function pointers to its API functions during driver initialization. These structures are placed into the RAM section in initialization level order.



7.7.2 Standard Drivers

Device drivers which are present on all supported board configurations are listed below.

- **Interrupt controller**: This device driver is used by the kernel's interrupt management subsystem.
- **Timer**: This device driver is used by the kernel's system clock and hardware clock subsystem.
- **Serial communication**: This device driver is used by the kernel's system console subsystem.
- **Entropy**: This device driver provides a source of entropy numbers for the random number generator subsystem.

Important: Use the *random API functions* for random values. *Entropy functions* should not be directly used as a random number generator source as some hardware implementations are designed to be an entropy seed source for random number generators and will not provide cryptographically secure random number streams.

7.7.3 Synchronous Calls

Zephyr provides a set of device drivers for multiple boards. Each driver should support an interrupt-based implementation, rather than polling, unless the specific hardware does not provide any interrupt.

High-level calls accessed through device-specific APIs, such as `i2c.h` or `spi.h`, are usually intended as synchronous. Thus, these calls should be blocking.

7.7.4 Driver APIs

The following APIs for device drivers are provided by `device.h`. The APIs are intended for use in device drivers only and should not be used in applications.

`DEVICE_DEFINE()` Create device object and related data structures including setting it up for boot-time initialization.

`DEVICE_NAME_GET()` Converts a device identifier to the global identifier for a device object.

`DEVICE_GET()` Obtain a pointer to a device object by name.

`DEVICE_DECLARE()` Declare a device object. Use this when you need a forward reference to a device that has not yet been defined.

7.7.5 Driver Data Structures

The device initialization macros populate some data structures at build time which are split into read-only and runtime-mutable parts. At a high level we have:

```
struct device {
      const char *name;
      const void *config;
      const void *api;
      void * const data;
};
```

The `config` member is for read-only configuration data set at build time. For example, base memory mapped IO addresses, IRQ line numbers, or other fixed physical characteristics of the device. This is the `config` pointer passed to `DEVICE_DEFINE()` and related macros.

The `data` struct is kept in RAM, and is used by the driver for per-instance runtime housekeeping. For example, it may contain reference counts, semaphores, scratch buffers, etc.

The `api` struct maps generic subsystem APIs to the device-specific implementations in the driver. It is typically read-only and populated at build time. The next section describes this in more detail.

7.7.6 Subsystems and API Structures

Most drivers will be implementing a device-independent subsystem API. Applications can simply program to that generic API, and application code is not specific to any particular driver implementation.

A subsystem API definition typically looks like this:

```
typedef int (*subsystem_do_this_t)(const struct device *dev, int foo, int bar);
typedef void (*subsystem_do_that_t)(const struct device *dev, void *baz);

struct subsystem_api {
      subsystem_do_this_t do_this;
      subsystem_do_that_t do_that;
```

(continues on next page)

(continued from previous page)

```
};

static inline int subsystem_do_this(const struct device *dev, int foo, int bar)
{
      struct subsystem_api *api;

      api = (struct subsystem_api *)dev->api;
      return api->do_this(dev, foo, bar);
}

static inline void subsystem_do_that(const struct device *dev, void *baz)
{
      struct subsystem_api *api;

      api = (struct subsystem_api *)dev->api;
      api->do_that(dev, baz);
}
```

A driver implementing a particular subsystem will define the real implementation of these APIs, and populate an instance of subsystem_api structure:

```
static int my_driver_do_this(const struct device *dev, int foo, int bar)
{
      ...
}

static void my_driver_do_that(const struct device *dev, void *baz)
{
      ...
}

static struct subsystem_api my_driver_api_funcs = {
      .do_this = my_driver_do_this,
      .do_that = my_driver_do_that
};
```

The driver would then pass `my_driver_api_funcs` as the `api` argument to `DEVICE_DEFINE()`.

Note: Since pointers to the API functions are referenced in the `api` struct, they will always be included in the binary even if unused; `gc-sections` linker option will always see at least one reference to them. Providing for link-time size optimizations with driver APIs in most cases requires that the optional feature be controlled by a Kconfig option.

7.7.7 Device-Specific API Extensions

Some devices can be cast as an instance of a driver subsystem such as GPIO, but provide additional functionality that cannot be exposed through the standard API. These devices combine subsystem operations with device-specific APIs, described in a device-specific header.

A device-specific API definition typically looks like this:

```
#include <drivers/subsystem.h>

/* When extensions need not be invoked from user mode threads */
int specific_do_that(const struct device *dev, int foo);
```

(continues on next page)

(continued from previous page)

```
/* When extensions must be invokable from user mode threads */
__syscall int specific_from_user(const struct device *dev, int bar);

/* Only needed when extensions include syscalls */
#include <syscalls/specific.h>
```

A driver implementing extensions to the subsystem will define the real implementation of both the subsystem API and the specific APIs:

```
static int generic_do_this(const struct device *dev, void *arg)
{
   ...
}

static struct generic_api api {
   ...
   .do_this = generic_do_this,
   ...
};

/* supervisor-only API is globally visible */
int specific_do_that(const struct device *dev, int foo)
{
   ...
}

/* syscall API passes through a translation */
int z_impl_specific_from_user(const struct device *dev, int bar)
{
   ...
}

#ifdef CONFIG_USERSPACE

#include <syscall_handler.h>

int z_vrfy_specific_from_user(const struct device *dev, int bar)
{
    Z_OOPS(Z_SYSCALL_SPECIFIC_DRIVER(dev, K_OBJ_DRIVER_GENERIC, &api));
    return z_impl_specific_do_that(dev, bar)
}

#include <syscalls/specific_from_user_mrsh.c>

#endif /* CONFIG_USERSPACE */
```

Applications use the device through both the subsystem and specific APIs.

Note: Public API for device-specific extensions should be prefixed with the compatible for the device to which it applies. For example, if adding special functions to support the Maxim DS3231 the identifier fragment `specific` in the examples above would be `maxim_ds3231`.

7.7.8 Single Driver, Multiple Instances

Some drivers may be instantiated multiple times in a given system. For example there can be multiple GPIO banks, or multiple UARTS. Each instance of the driver will have a different `config` struct and `data` struct.

Configuring interrupts for multiple drivers instances is a special case. If each instance needs to configure a different interrupt line, this can be accomplished through the use of per-instance configuration functions, since the parameters to `IRQ_CONNECT()` need to be resolvable at build time.

For example, let's say we need to configure two instances of `my_driver`, each with a different interrupt line. In `drivers/subsystem/subsystem_my_driver.h`:

```
typedef void (*my_driver_config_irq_t)(const struct device *dev);

struct my_driver_config {
	DEVICE_MMIO_ROM;
	my_driver_config_irq_t config_func;
};
```

In the implementation of the common init function:

```
void my_driver_isr(const struct device *dev)
{
	/* Handle interrupt */
	...
}

int my_driver_init(const struct device *dev)
{
	const struct my_driver_config *config = dev->config;

	DEVICE_MMIO_MAP(dev, K_MEM_CACHE_NONE);

	/* Do other initialization stuff */
	...

	config->config_func(dev);

	return 0;
}
```

Then when the particular instance is declared:

```
#if CONFIG_MY_DRIVER_0

DEVICE_DECLARE(my_driver_0);

static void my_driver_config_irq_0(void)
{
	IRQ_CONNECT(MY_DRIVER_0_IRQ, MY_DRIVER_0_PRI, my_driver_isr,
		    DEVICE_GET(my_driver_0), MY_DRIVER_0_FLAGS);
}

const static struct my_driver_config my_driver_config_0 = {
	DEVICE_MMIO_ROM_INIT(DT_DRV_INST(0)),
	.config_func = my_driver_config_irq_0
}
```

(continues on next page)

(continued from previous page)

```
static struct my_data_0;

DEVICE_DEFINE(my_driver_0, MY_DRIVER_0_NAME, my_driver_init,
              NULL, &my_data_0, &my_driver_config_0,
              POST_KERNEL, MY_DRIVER_0_PRIORITY, &my_api_funcs);

#endif /* CONFIG_MY_DRIVER_0 */
```

Note the use of DEVICE_DECLARE() to avoid a circular dependency on providing the IRQ handler argument and the definition of the device itself.

7.7.9 Initialization Levels

Drivers may depend on other drivers being initialized first, or require the use of kernel services. *DEVICE_DEFINE()* and related APIs allow the user to specify at what time during the boot sequence the init function will be executed. Any driver will specify one of four initialization levels:

PRE_KERNEL_1 Used for devices that have no dependencies, such as those that rely solely on hardware present in the processor/SOC. These devices cannot use any kernel services during configuration, since the kernel services are not yet available. The interrupt subsystem will be configured however so it's OK to set up interrupts. Init functions at this level run on the interrupt stack.

PRE_KERNEL_2 Used for devices that rely on the initialization of devices initialized as part of the PRE_KERNEL_1 level. These devices cannot use any kernel services during configuration, since the kernel services are not yet available. Init functions at this level run on the interrupt stack.

POST_KERNEL Used for devices that require kernel services during configuration. Init functions at this level run in context of the kernel main task.

APPLICATION Used for application components (i.e. non-kernel components) that need automatic configuration. These devices can use all services provided by the kernel during configuration. Init functions at this level run on the kernel main task.

Within each initialization level you may specify a priority level, relative to other devices in the same initialization level. The priority level is specified as an integer value in the range 0 to 99; lower values indicate earlier initialization. The priority level must be a decimal integer literal without leading zeroes or sign (e.g. 32), or an equivalent symbolic name (e.g. \#define MY_INIT_PRIO 32); symbolic expressions are *not* permitted (e.g. CONFIG_KERNEL_INIT_PRIORITY_DEFAULT + 5).

Drivers and other system utilities can determine whether startup is still in pre-kernel states by using the *k_is_pre_kernel()* function.

7.7.10 System Drivers

In some cases you may just need to run a function at boot. Special SYS_* macros exist that map to DEVICE_DEFINE() calls. For SYS_INIT() there are no config or runtime data structures and there isn't a way to later get a device pointer by name. The same policies for initialization level and priority apply.

For SYS_DEVICE_DEFINE() you can obtain pointers by name, see *power management* section.

SYS_INIT() Run an initialization function at boot at specified priority.

SYS_DEVICE_DEFINE() Like *DEVICE_DEFINE()* without an API table and constructing the device name from the init function name.

7.7.11 Error handling

In general, it's best to use `__ASSERT()` macros instead of propagating return values unless the failure is expected to occur during the normal course of operation (such as a storage device full). Bad parameters, programming errors, consistency checks, pathological/unrecoverable failures, etc., should be handled by assertions.

When it is appropriate to return error conditions for the caller to check, 0 should be returned on success and a POSIX `errno.h` code returned on failure. See https://github.com/zephyrproject-rtos/zephyr/wiki/Naming-Conventions#return-codes for details about this.

7.7.12 Memory Mapping

On some systems, the linear address of peripheral memory-mapped I/O (MMIO) regions cannot be known at build time:

- The I/O ranges must be probed at runtime from the bus, such as with PCI express
- A memory management unit (MMU) is active, and the physical address of the MMIO range must be mapped into the page tables at some virtual memory location determined by the kernel.

These systems must maintain storage for the MMIO range within RAM and establish the mapping within the driver's init function. Other systems do not care about this and can use MMIO physical addresses directly from DTS and do not need any RAM-based storage for it.

For drivers that may need to deal with this situation, a set of APIs under the DEVICE_MMIO scope are defined, along with a mapping function `device_map()`.

Device Model Drivers with one MMIO region

The simplest case is for drivers which need to maintain one MMIO region. These drivers will need to use the `DEVICE_MMIO_ROM` and `DEVICE_MMIO_RAM` macros in the definitions for their `config_info` and `driver_data` structures, with initialization of the `config_info` from DTS using `DEVICE_MMIO_ROM_INIT`. A call to `DEVICE_MMIO_MAP()` is made within the init function:

```
struct my_driver_config {
   DEVICE_MMIO_ROM; /* Must be first */
   ...
}

struct my_driver_dev_data {
   DEVICE_MMIO_RAM; /* Must be first */
   ...
}

const static struct my_driver_config my_driver_config_0 = {
   DEVICE_MMIO_ROM_INIT(DT_DRV_INST(...)),
   ...
}

int my_driver_init(const struct device *dev)
{
   ...
   DEVICE_MMIO_MAP(dev, K_MEM_CACHE_NONE);
   ...
}

int my_driver_some_function(const struct device *dev)
```

(continues on next page)

(continued from previous page)

```
{
   ...
   /* Write some data to the MMIO region */
   sys_write32(0xDEADBEEF, DEVICE_MMIO_GET(dev));
   ...
}
```

The particular expansion of these macros depends on configuration. On a device with no MMU or PCI-e, `DEVICE_MMIO_MAP` and `DEVICE_MMIO_RAM` expand to nothing.

Device Model Drivers with multiple MMIO regions

Some drivers may have multiple MMIO regions. In addition, some drivers may already be implementing a form of inheritance which requires some other data to be placed first in the `config_info` and `driver_data` structures.

This can be managed with the `DEVICE_MMIO_NAMED` variant macros. These require that `DEV_CFG()` and `DEV_DATA()` macros be defined to obtain a properly typed pointer to the driver's config_info or dev_data structs. For example:

```
struct my_driver_config {
   ...
     DEVICE_MMIO_NAMED_ROM(corge);
     DEVICE_MMIO_NAMED_ROM(grault);
   ...
}

struct my_driver_dev_data {
        ...
     DEVICE_MMIO_NAMED_RAM(corge);
     DEVICE_MMIO_NAMED_RAM(grault);
      ...
}

#define DEV_CFG(_dev) \
   ((const struct my_driver_config *)((_dev)->config))

#define DEV_DATA(_dev) \
   ((struct my_driver_dev_data *)((_dev)->data))

const static struct my_driver_config my_driver_config_0 = {
   ...
   DEVICE_MMIO_NAMED_ROM_INIT(corge, DT_DRV_INST(...)),
   DEVICE_MMIO_NAMED_ROM_INIT(grault, DT_DRV_INST(...)),
   ...
}

int my_driver_init(const struct device *dev)
{
   ...
   DEVICE_MMIO_NAMED_MAP(dev, corge, K_MEM_CACHE_NONE);
   DEVICE_MMIO_NAMED_MAP(dev, grault, K_MEM_CACHE_NONE);
   ...
}

int my_driver_some_function(const struct device *dev)
```

(continues on next page)

(continued from previous page)

```
{
   ...
   /* Write some data to the MMIO regions */
   sys_write32(0xDEADBEEF, DEVICE_MMIO_GET(dev, grault));
   sys_write32(0xF0CCAC1A, DEVICE_MMIO_GET(dev, corge));
   ...
}
```

Drivers that do not use Zephyr Device Model

Some drivers or driver-like code may not user Zephyr's device model, and alternative storage must be arranged for the MMIO data. An example of this are timer drivers, or interrupt controller code.

This can be managed with the `DEVICE_MMIO_TOPLEVEL` set of macros, for example:

```
DEVICE_MMIO_TOPLEVEL_STATIC(my_regs, DT_DRV_INST(..));

void some_init_code(...)
{
   ...
   DEVICE_MMIO_TOPLEVEL_MAP(my_regs, K_MEM_CACHE_NONE);
   ...
}

void some_function(...)
   ...
   sys_write32(DEVICE_MMIO_TOPLEVEL_GET(my_regs), 0xDEADBEEF);
   ...
}
```

Drivers that do not use DTS

Some drivers may not obtain the MMIO physical address from DTS, such as is the case with PCI-E. In this case the `device_map()` function may be used directly:

```
void some_init_code(...)
{
   ...
   struct pcie_mbar mbar;
   bool bar_found = pcie_get_mbar(bdf, index, &mbar);

   device_map(DEVICE_MMIO_RAM_PTR(dev), mbar.phys_addr, mbar.size, K_MEM_CACHE_NONE);
   ...
}
```

For these cases, DEVICE_MMIO_ROM directives may be omitted.

7.7.13 API Reference

group `device_model`

Device Model APIs.

Defines

DEVICE_HANDLE_SEP

Flag value used in lists of device handles to separate distinct groups.

This is the minimum value for the device_handle_t type.

DEVICE_HANDLE_ENDS

Flag value used in lists of device handles to indicate the end of the list.

This is the maximum value for the device_handle_t type.

DEVICE_HANDLE_NULL

Flag value used to identify an unknown device.

DEVICE_NAME_GET(name)

Expands to the full name of a global device object.

Return the full name of a device object symbol created by *DEVICE_DEFINE()*, using the dev_name provided to *DEVICE_DEFINE()*.

It is meant to be used for declaring extern symbols pointing on device objects before using the DEVICE_GET macro to get the device object.

Parameters

- `name` – The same as dev_name provided to *DEVICE_DEFINE()*

Returns The expanded name of the device object created by *DEVICE_DEFINE()*

SYS_DEVICE_DEFINE(drv_name, init_fn, pm_control_fn, level, prio)

Run an initialization function at boot at specified priority, and define device PM control function.

Invokes *DEVICE_DEFINE()* with no power management support (`pm_control_fn`), no API (`api_ptr`), and a device name derived from the `init_fn` name (`dev_name`).

DEVICE_DEFINE(dev_name, drv_name, init_fn, pm_control_fn, data_ptr, cfg_ptr, level, prio, api_ptr)

Create device object and set it up for boot time initialization, with the option to pm_control. In case of Device Idle Power Management is enabled, make sure the device is in suspended state after initialization.

This macro defines a device object that is automatically configured by the kernel during system initialization. Note that devices set up with this macro will not be accessible from user mode since the API is not specified;

Parameters

- `dev_name` – Device name. This must be less than Z_DEVICE_MAX_NAME_LEN characters (including terminating NUL) in order to be looked up from user mode with *device_get_binding()*.
- `drv_name` – The name this instance of the driver exposes to the system.
- `init_fn` – Address to the init function of the driver.
- `pm_control_fn` – Pointer to pm_control function. Can be NULL if not implemented.
- `data_ptr` – Pointer to the device's private data.
- `cfg_ptr` – The address to the structure containing the configuration information for this instance of the driver.

- `level` – The initialization level. See *SYS_INIT()* for details.
- `prio` – Priority within the selected initialization level. See *SYS_INIT()* for details.
- `api_ptr` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

`DEVICE_DT_NAME(node_id)`

Return a string name for a devicetree node.

This macro returns a string literal usable as a device name from a devicetree node. If the node has a "label" property, its value is returned. Otherwise, the node's full "node-name@@unit-address" name is returned.

Parameters

- `node_id` – The devicetree node identifier.

`DEVICE_DT_DEFINE(node_id, init_fn, pm_control_fn, data_ptr, cfg_ptr, level, prio, api_ptr, ...)`

Like DEVICE_DEFINE but taking metadata from a devicetree node.

This macro defines a device object that is automatically configured by the kernel during system initialization. The device object name is derived from the node identifier (encoding the devicetree path to the node), and the driver name is from the `label` property of the devicetree node.

The device is declared with extern visibility, so device objects defined through this API can be obtained directly through *DEVICE_DT_GET()* using `node_id`. Before using the pointer the referenced object should be checked using *device_is_ready()*.

Parameters

- `node_id` – The devicetree node identifier.
- `init_fn` – Address to the init function of the driver.
- `pm_control_fn` – Pointer to pm_control function. Can be NULL if not implemented.
- `data_ptr` – Pointer to the device's private data.
- `cfg_ptr` – The address to the structure containing the configuration information for this instance of the driver.
- `level` – The initialization level. See *SYS_INIT()* for details.
- `prio` – Priority within the selected initialization level. See *SYS_INIT()* for details.
- `api_ptr` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.

`DEVICE_DT_INST_DEFINE(inst, ...)`

Like DEVICE_DT_DEFINE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- `inst` – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to DEVICE_DT_DEFINE.
- `...` – other parameters as expected by DEVICE_DT_DEFINE.

`DEVICE_DT_NAME_GET(node_id)`

The name of the struct device object for `node_id`.

Return the full name of a device object symbol created by *DEVICE_DT_DEFINE()*, using the dev_name derived from `node_id`

It is meant to be used for declaring extern symbols pointing on device objects before using the DEVICE_DT_GET macro to get the device object.

Parameters

- `node_id` – The same as node_id provided to *DEVICE_DT_DEFINE()*

Returns The expanded name of the device object created by *DEVICE_DT_DEFINE()*

`DEVICE_DT_GET(node_id)`

Obtain a pointer to a device object by `node_id`.

Return the address of a device object created by DEVICE_DT_INIT(), using the dev_name derived from `node_id`

Parameters

- `node_id` – The same as node_id provided to *DEVICE_DT_DEFINE()*

Returns A pointer to the device object created by *DEVICE_DT_DEFINE()*

`DEVICE_DT_INST_GET(inst)`

Obtain a pointer to a device object for an instance of a DT_DRV_COMPAT compatible.

Parameters

- `inst` – instance number

`DEVICE_DT_GET_ANY(compat)`

Obtain a pointer to a device object by devicetree compatible.

If any enabled devicetree node has the given compatible and a device object was created from it, this returns that device.

If there no such devices, this returns NULL.

If there are multiple, this returns an arbitrary one.

If this returns non-NULL, the device must be checked for readiness before use, e.g. with *device_is_ready()*.

Parameters

- `compat` – lowercase-and-underscores devicetree compatible

Returns a pointer to a device, or NULL

`DEVICE_DT_GET_ONE(compat)`

Obtain a pointer to a device object by devicetree compatible.

If any enabled devicetree node has the given compatible and a device object was created from it, this returns that device.

If there no such devices, this throws a compilation error.

If there are multiple, this returns an arbitrary one.

If this returns non-NULL, the device must be checked for readiness before use, e.g. with *device_is_ready()*.

Parameters

- `compat` – lowercase-and-underscores devicetree compatible

Returns a pointer to a device

`DEVICE_GET(name)`

Obtain a pointer to a device object by name.

Return the address of a device object created by *DEVICE_DEFINE()*, using the dev_name provided to *DEVICE_DEFINE()*.

Parameters

- `name` – The same as dev_name provided to *DEVICE_DEFINE()*

Returns A pointer to the device object created by *DEVICE_DEFINE()*

`DEVICE_DECLARE(name)`

Declare a static device object.

This macro can be used at the top-level to declare a device, such that *DEVICE_GET()* may be used before the full declaration in *DEVICE_DEFINE()*.

This is often useful when configuring interrupts statically in a device's init or per-instance config function, as the init function itself is required by *DEVICE_DEFINE()* and use of *DEVICE_GET()* inside it creates a circular dependency.

Parameters

- `name` – Device name

`SYS_INIT(_init_fn, _level, _prio)`

Run an initialization function at boot at specified priority.

This macro lets you run a function at system boot.

Parameters

- `_init_fn` – Pointer to the boot function to run
- `_level` – The initialization level at which configuration occurs. Must be one of the following symbols, which are listed in the order they are performed by the kernel:
  - PRE_KERNEL_1: Used for initialization objects that have no dependencies, such as those that rely solely on hardware present in the processor/SOC. These objects cannot use any kernel services during configuration, since they are not yet available.
  - PRE_KERNEL_2: Used for initialization objects that rely on objects initialized as part of the PRE_KERNEL_1 level. These objects cannot use any kernel services during configuration, since they are not yet available.
  - POST_KERNEL: Used for initialization objects that require kernel services during configuration.
  - POST_KERNEL_SMP: Used for initialization objects that require kernel services during configuration after SMP initialization.
  - APPLICATION: Used for application components (i.e. non-kernel components) that need automatic configuration. These objects can use all services provided by the kernel during configuration.
- `_prio` – The initialization priority of the object, relative to other objects of the same initialization level. Specified as an integer value in the range 0 to 99; lower values indicate earlier initialization. Must be a decimal integer literal without leading zeroes or sign (e.g. 32), or an equivalent symbolic name (e.g. #define MY_INIT_PRIO 32); symbolic expressions are *not* permitted (e.g. CONFIG_KERNEL_INIT_PRIORITY_DEFAULT + 5).

Typedefs

typedef int16_t `device_handle_t`

Type used to represent devices and functions.

The extreme values and zero have special significance. Negative values identify functionality that does not correspond to a Zephyr device, such as the system clock or a *SYS_INIT()* function.

typedef int (*`device_visitor_callback_t`)(const struct *device* *dev, void *context)

Prototype for functions used when iterating over a set of devices.

Such a function may be used in API that identifies a set of devices and provides a visitor API supporting caller-specific interaction with each device in the set.

The visit is said to succeed if the visitor returns a non-negative value.

Param dev a device in the set being iterated

Param context state used to support the visitor function

Return A non-negative number to allow walking to continue, and a negative error code to case the iteration to stop.

Functions

static inline *device_handle_t* `device_handle_get`(const struct *device* *dev)

Get the handle for a given device.

Parameters

- `dev` – the device for which a handle is desired.

Returns the handle for the device, or DEVICE_HANDLE_NULL if the device does not have an associated handle.

static inline const struct *device* *`device_from_handle`(*device_handle_t* dev_handle)

Get the device corresponding to a handle.

Parameters

- `dev_handle` – the device handle

Returns the device that has that handle, or a null pointer if `dev_handle` does not identify a device.

static inline const *device_handle_t* *`device_required_handles_get`(const struct *device* *dev, size_t *count)

Get the set of handles for devicetree dependencies of this device.

These are the device dependencies inferred from devicetree.

Parameters

- `dev` – the device for which dependencies are desired.
- `count` – pointer to a place to store the number of devices provided at the returned pointer. The value is not set if the call returns a null pointer. The value may be set to zero.

Returns a pointer to a sequence of `*count` device handles, or a null pointer if `dh` does not provide dependency information.

int `device_required_foreach`(const struct *device* *dev, *device_visitor_callback_t* visitor_cb, void *context)

Visit every device that `dev` directly requires.

Zephyr maintains information about which devices are directly required by another device; for example an I2C-based sensor driver will require an I2C controller for communication. Required devices can derive from statically-defined devicetree relationships or dependencies registered at runtime.

This API supports operating on the set of required devices. Example uses include making sure required devices are ready before the requiring device is used, and releasing them when the requiring device is no longer needed.

There is no guarantee on the order in which required devices are visited.

If the `visitor` function returns a negative value iteration is halted, and the returned value from the visitor is returned from this function.

Note: This API is not available to unprivileged threads.

Parameters

- `dev` – a device of interest. The devices that this device depends on will be used as the set of devices to visit. This parameter must not be null.
- `visitor_cb` – the function that should be invoked on each device in the dependency set. This parameter must not be null.
- `context` – state that is passed through to the visitor function. This parameter may be null if `visitor` tolerates a null `context`.

Returns The number of devices that were visited if all visits succeed, or the negative value returned from the first visit that did not succeed.

const struct *device* *`device_get_binding`(const char *name)

Retrieve the device structure for a driver by name.

Device objects are created via the *DEVICE_DEFINE()* macro and placed in memory by the linker. If a driver needs to bind to another driver it can use this function to retrieve the device structure of the lower level driver by the name the driver exposes to the system.

Parameters

- `name` – device name to search for. A null pointer, or a pointer to an empty string, will cause NULL to be returned.

Returns pointer to device structure; NULL if not found or cannot be used.

int `device_usable_check`(const struct *device* *dev)

Determine whether a device is ready for use.

This checks whether a device can be used, returning 0 if it can, and distinct error values that identify the reason if it cannot.

Return values

- `0` – if the device is usable.
- `-ENODEV` – if the device has not been initialized, the device pointer is NULL or the initialization failed.
- `other` – negative error codes to indicate additional conditions that make the device unusable.

static inline bool `device_is_ready`(const struct *device* *dev)

Verify that a device is ready for use.

Indicates whether the provided device pointer is for a device known to be in a state where it can be used with its standard API.

This can be used with device pointers captured from *DEVICE_DT_GET()*, which does not include the readiness checks of *device_get_binding()*. At minimum this means that the device has been successfully initialized, but it may take on further conditions (e.g. is not powered down).

Parameters

- `dev` – pointer to the device in question.

Return values

- `true` – if the device is ready for use.
- `false` – if the device is not ready for use or if a NULL device pointer is passed as argument.

struct `device_state`

#include <device.h> Runtime device dynamic structure (in RAM) per driver instance.

Fields in this are expected to be default-initialized to zero. The kernel driver infrastructure and driver access functions are responsible for ensuring that any non-zero initialization is done before they are accessed.

Public Members

unsigned int `init_res`

Non-negative result of initializing the device.

The absolute value returned when the device initialization function was invoked, or `UINT8_MAX` if the value exceeds an 8-bit integer. If initialized is also set, a zero value indicates initialization succeeded.

bool `initialized`

Indicates the device initialization function has been invoked.

struct `device`

#include <device.h> Runtime device structure (in ROM) per driver instance.

Public Members

const char *`name`

Name of the device instance

const void *`config`

Address of device instance config information

const void *`api`

Address of the API structure exposed by the device instance

struct *device_state* *const `state`

Address of the common device state

void *const `data`

Address of the device instance private data

const *device_handle_t* *const `handles`

optional pointer to handles associated with the device.

This encodes a sequence of sets of device handles that have some relationship to this node. The individual sets are extracted with dedicated API, such as *device_required_handles_get()*.

pm_device_control_callback_t `pm_control`

Power Management function

struct *pm_device* *const `pm`

Pointer to device instance power management data

7.8 Display Interface

7.8.1 API Reference

Generic Display Interface

group `display_interface`

Display Interface.

Typedefs

typedef int (*`display_blanking_on_api`)(const struct *device* *dev)

Callback API to turn on display blanking See *display_blanking_on()* for argument description.

typedef int (*`display_blanking_off_api`)(const struct *device* *dev)

Callback API to turn off display blanking See *display_blanking_off()* for argument description.

typedef int (*`display_write_api`)(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, const void *buf)

Callback API for writing data to the display See *display_write()* for argument description.

typedef int (*`display_read_api`)(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, void *buf)

Callback API for reading data from the display See *display_read()* for argument description.

typedef void *(*`display_get_framebuffer_api`)(const struct *device* *dev)

Callback API to get framebuffer pointer See *display_get_framebuffer()* for argument description.

typedef int (*`display_set_brightness_api`)(const struct *device* *dev, const uint8_t brightness)

Callback API to set display brightness See *display_set_brightness()* for argument description.

typedef int (*`display_set_contrast_api`)(const struct *device* *dev, const uint8_t contrast)

Callback API to set display contrast See *display_set_contrast()* for argument description.

typedef void (*`display_get_capabilities_api`)(const struct *device* *dev, struct *display_capabilities* *capabilities)

Callback API to get display capabilities See *display_get_capabilities()* for argument description.

typedef int (*`display_set_pixel_format_api`)(const struct *device* *dev, const enum *display_pixel_format* pixel_format)

Callback API to set pixel format used by the display See *display_set_pixel_format()* for argument description.

typedef int (*`display_set_orientation_api`)(const struct *device* *dev, const enum *display_orientation* orientation)

Callback API to set orientation used by the display See *display_set_orientation()* for argument description.

Enums

enum `display_pixel_format`

Display pixel formats.

Display pixel format enumeration.

In case a pixel format consists out of multiple bytes the byte order is big endian.

Values:

enumerator `PIXEL_FORMAT_RGB_888` = *BIT*(0)

enumerator `PIXEL_FORMAT_MONO01` = *BIT*(1)

enumerator `PIXEL_FORMAT_MONO10` = *BIT*(2)

enumerator `PIXEL_FORMAT_ARGB_8888` = *BIT*(3)

enumerator `PIXEL_FORMAT_RGB_565` = *BIT*(4)

enumerator `PIXEL_FORMAT_BGR_565` = *BIT*(5)

enum `display_screen_info`

Values:

enumerator `SCREEN_INFO_MONO_VTILED` = *BIT*(0)

If selected, one octet represents 8 pixels ordered vertically, otherwise ordered horizontally.

enumerator `SCREEN_INFO_MONO_MSB_FIRST` = *BIT*(1)

If selected, the MSB represents the first pixel, otherwise MSB represents the last pixel.

enumerator `SCREEN_INFO_EPD` = *BIT*(2)

Electrophoretic Display.

enumerator SCREEN_INFO_DOUBLE_BUFFER = *BIT*(3)

Screen has two alternating ram buffers

enumerator SCREEN_INFO_X_ALIGNMENT_WIDTH = *BIT*(4)

Screen has x alignment constrained to width.

enum display_orientation

Enumeration with possible display orientation.

Values:

enumerator DISPLAY_ORIENTATION_NORMAL

enumerator DISPLAY_ORIENTATION_ROTATED_90

enumerator DISPLAY_ORIENTATION_ROTATED_180

enumerator DISPLAY_ORIENTATION_ROTATED_270

Functions

static inline int display_write(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, const void *buf)

Write data to display.

Parameters

- dev – Pointer to device structure
- x – x Coordinate of the upper left corner where to write the buffer
- y – y Coordinate of the upper left corner where to write the buffer
- desc – Pointer to a structure describing the buffer layout
- buf – Pointer to buffer array

Return values 0 – on success else negative errno code.

static inline int display_read(const struct *device* *dev, const uint16_t x, const uint16_t y, const struct *display_buffer_descriptor* *desc, void *buf)

Read data from display.

Parameters

- dev – Pointer to device structure
- x – x Coordinate of the upper left corner where to read from
- y – y Coordinate of the upper left corner where to read from
- desc – Pointer to a structure describing the buffer layout
- buf – Pointer to buffer array

Return values 0 – on success else negative errno code.

static inline void *display_get_framebuffer(const struct *device* *dev)

Get pointer to framebuffer for direct access.

Parameters

- dev – Pointer to device structure

Return values Pointer – to frame buffer or NULL if direct framebuffer access is not supported

static inline int display_blanking_on(const struct *device* *dev)

Turn display blanking on.

This function blanks the complete display. The content of the frame buffer will be retained while blanking is enabled and the frame buffer will be accessible for read and write operations.

In case backlight control is supported by the driver the backlight is turned off. The backlight configuration is retained and accessible for configuration.

In case the driver supports display blanking the initial state of the driver would be the same as if this function was called.

Parameters

- dev – Pointer to device structure

Return values 0 – on success else negative errno code.

static inline int display_blanking_off(const struct *device* *dev)

Turn display blanking off.

Restore the frame buffer content to the display. In case backlight control is supported by the driver the backlight configuration is restored.

Parameters

- dev – Pointer to device structure

Return values 0 – on success else negative errno code.

static inline int display_set_brightness(const struct *device* *dev, uint8_t brightness)

Set the brightness of the display.

Set the brightness of the display in steps of 1/256, where 255 is full brightness and 0 is minimal.

Parameters

- dev – Pointer to device structure
- brightness – Brightness in steps of 1/256

Return values 0 – on success else negative errno code.

static inline int display_set_contrast(const struct *device* *dev, uint8_t contrast)

Set the contrast of the display.

Set the contrast of the display in steps of 1/256, where 255 is maximum difference and 0 is minimal.

Parameters

- dev – Pointer to device structure
- contrast – Contrast in steps of 1/256

Return values 0 – on success else negative errno code.

static inline void `display_get_capabilities`(const struct *device* *dev, struct *display_capabilities* *capabilities)

Get display capabilities.

Parameters

- `dev` – Pointer to device structure
- `capabilities` – Pointer to capabilities structure to populate

static inline int `display_set_pixel_format`(const struct *device* *dev, const enum *display_pixel_format* pixel_format)

Set pixel format used by the display.

Parameters

- `dev` – Pointer to device structure
- `pixel_format` – Pixel format to be used by display

Return values `0` – on success else negative errno code.

static inline int `display_set_orientation`(const struct *device* *dev, const enum *display_orientation* orientation)

Set display orientation.

Parameters

- `dev` – Pointer to device structure
- `orientation` – Orientation to be used by display

Return values `0` – on success else negative errno code.

struct `display_capabilities`

#include <display.h> Structure holding display capabilities.

Public Members

uint16_t `x_resolution`

Display resolution in the X direction

uint16_t `y_resolution`

Display resolution in the Y direction

uint32_t `supported_pixel_formats`

Bitwise or of pixel formats supported by the display

uint32_t `screen_info`

Information about display panel

enum *display_pixel_format* `current_pixel_format`

Currently active pixel format for the display

enum *display_orientation* `current_orientation`

Current display orientation

struct `display_buffer_descriptor`

#include <display.h> Structure to describe display data buffer layout.

Public Members

uint32_t `buf_size`
Data buffer size in bytes

uint16_t `width`
Data buffer row width in pixels

uint16_t `height`
Data buffer column height in pixels

uint16_t `pitch`
Number of pixels between consecutive rows in the data buffer

struct `display_driver_api`
#include <display.h> Display driver API API which a display driver should expose.

Grove LCD Display

group `grove_display`
Grove display APIs.

Defines

`GROVE_LCD_NAME`

`GLCD_DS_DISPLAY_ON`

`GLCD_DS_DISPLAY_OFF`

`GLCD_DS_CURSOR_ON`

`GLCD_DS_CURSOR_OFF`

`GLCD_DS_BLINK_ON`

`GLCD_DS_BLINK_OFF`

`GLCD_IS_SHIFT_INCREMENT`

`GLCD_IS_SHIFT_DECREMENT`

`GLCD_IS_ENTRY_LEFT`

`GLCD_IS_ENTRY_RIGHT`

GLCD_FS_8BIT_MODE

GLCD_FS_ROWS_2

GLCD_FS_ROWS_1

GLCD_FS_DOT_SIZE_BIG

GLCD_FS_DOT_SIZE_LITTLE

GROVE_RGB_WHITE

GROVE_RGB_RED

GROVE_RGB_GREEN

GROVE_RGB_BLUE

Functions

void glcd_print(const struct *device* *port, char *data, uint32_t size)

Send text to the screen.

Parameters

- port – Pointer to device structure for driver instance.
- data – the ASCII text to display
- size – the length of the text in bytes

void glcd_cursor_pos_set(const struct *device* *port, uint8_t col, uint8_t row)

Set text cursor position for next additions.

Parameters

- port – Pointer to device structure for driver instance.
- col – the column for the cursor to be moved to (0-15)
- row – the row it should be moved to (0 or 1)

void glcd_clear(const struct *device* *port)

Clear the current display.

Parameters

- port – Pointer to device structure for driver instance.

void glcd_display_state_set(const struct *device* *port, uint8_t opt)

Function to change the display state.

This function provides the user the ability to change the state of the display as per needed. Controlling things like powering on or off the screen, the option to display the cursor or not, and the ability to blink the cursor.

Parameters

- port – Pointer to device structure for driver instance.

- `opt` – An 8bit bitmask of GLCD_DS_* options.

uint8_t `glcd_display_state_get`(const struct *device* *port)

return the display feature set associated with the device

Parameters

- `port` – the Grove LCD to get the display features set

Returns the display feature set associated with the device.

void `glcd_input_state_set`(const struct *device* *port, uint8_t opt)

Function to change the input state.

This function provides the user the ability to change the state of the text input. Controlling things like text entry from the left or right side, and how far to increment on new text

Parameters

- `port` – Pointer to device structure for driver instance.
- `opt` – A bitmask of GLCD_IS_* options

uint8_t `glcd_input_state_get`(const struct *device* *port)

return the input set associated with the device

Parameters

- `port` – the Grove LCD to get the input features set

Returns the input set associated with the device.

void `glcd_function_set`(const struct *device* *port, uint8_t opt)

Function to set the functional state of the display.

This function provides the user the ability to change the state of the display as per needed. Controlling things like the number of rows, dot size, and text display quality.

Parameters

- `port` – Pointer to device structure for driver instance.
- `opt` – A bitmask of GLCD_FS_* options

uint8_t `glcd_function_get`(const struct *device* *port)

return the function set associated with the device

Parameters

- `port` – the Grove LCD to get the functions set

Returns the function features set associated with the device.

void `glcd_color_select`(const struct *device* *port, uint8_t color)

Set LCD background to a predefined color.

Parameters

- `port` – Pointer to device structure for driver instance.
- `color` – One of the predefined color options

void `glcd_color_set`(const struct *device* *port, uint8_t r, uint8_t g, uint8_t b)

Set LCD background to custom RGB color value.

Parameters

- `port` – Pointer to device structure for driver instance.
- `r` – A numeric value for the red color (max is 255)

- `g` – A numeric value for the green color (max is 255)
- `b` – A numeric value for the blue color (max is 255)

int `glcd_initialize`(const struct *device* *port)

Initialize the Grove LCD panel.

Parameters

- `port` – Pointer to device structure for driver instance.

Returns Returns 0 if all passes

BBC micro:bit Display

group `mb_display`

BBC micro:bit display APIs.

Defines

`MB_IMAGE`(_rows...)

Generate an image object from a given array rows/columns.

This helper takes an array of 5 rows, each consisting of 5 0/1 values which correspond to the columns of that row. The value 0 means the pixel is disabled whereas a 1 means the pixel is enabled.

The pixels go from left to right and top to bottom, i.e. top-left corner is the first row's first value, top-right is the first rows last value, and bottom-right corner is the last value of the last (5th) row. As an example, the following would create a smiley face image:

Parameters

- `_rows` – Each of the 5 rows represented as a 5-value column array.

Returns Image bitmap that can be passed e.g. to *mb_display_image()*.

Enums

enum `mb_display_mode`

Display mode.

First 16 bits are reserved for modes, last 16 for flags.

Values:

enumerator `MB_DISPLAY_MODE_DEFAULT`

Default mode ("single" for images, "scroll" for text).

enumerator `MB_DISPLAY_MODE_SINGLE`

Display images sequentially, one at a time.

enumerator `MB_DISPLAY_MODE_SCROLL`

Display images by scrolling.

enumerator `MB_DISPLAY_FLAG_LOOP` = *BIT*(16)

Loop back to the beginning when reaching the last image.

Functions

struct mb_display *mb_display_get(void)

Get a pointer to the BBC micro:bit display object.

Returns Pointer to display object.

void mb_display_image(struct mb_display *disp, uint32_t mode, int32_t duration, const struct *mb_image* *img, uint8_t img_count)

Display one or more images on the BBC micro:bit LED display.

This function takes an array of one or more images and renders them sequentially on the micro:bit display. The call is asynchronous, i.e. the processing of the display happens in the background. If there is another image being displayed it will be canceled and the new one takes over.

Parameters

- disp – Display object.
- mode – One of the MB_DISPLAY_MODE_* options.
- duration – Duration how long to show each image (in milliseconds), or SYS_FOREVER_MS.
- img – Array of image bitmaps (struct *mb_image* objects).
- img_count – Number of images in 'img' array.

void mb_display_print(struct mb_display *disp, uint32_t mode, int32_t duration, const char *fmt, ...)

Print a string of characters on the BBC micro:bit LED display.

This function takes a printf-style format string and outputs it in a scrolling fashion to the display.

The call is asynchronous, i.e. the processing of the display happens in the background. If there is another image or string being displayed it will be canceled and the new one takes over.

Parameters

- disp – Display object.
- mode – One of the MB_DISPLAY_MODE_* options.
- duration – Duration how long to show each character (in milliseconds), or SYS_FOREVER_MS.
- fmt – printf-style format string
- ... – Optional list of format arguments.

void mb_display_stop(struct mb_display *disp)

Stop the ongoing display of an image.

Parameters

- disp – Display object.

struct mb_image

#include <mb_display.h> Representation of a BBC micro:bit display image.

This struct should normally not be used directly, rather created using the *MB_IMAGE()* macro.

Monochrome Character Framebuffer

group `monochrome_character_framebuffer`

Public Monochrome Character Framebuffer API.

Defines

`FONT_ENTRY_DEFINE(_name, _width, _height, _caps, _data, _fc, _lc)`

Macro for creating a font entry.

Parameters

- `_name` – Name of the font entry.
- `_width` – Width of the font in pixels
- `_height` – Height of the font in pixels.
- `_caps` – Font capabilities.
- `_data` – Raw data of the font.
- `_fc` – Character mapped to first font element.
- `_lc` – Character mapped to last font element.

Enums

enum `cfb_display_param`

Values:

enumerator `CFB_DISPLAY_HEIGH` = 0

enumerator `CFB_DISPLAY_WIDTH`

enumerator `CFB_DISPLAY_PPT`

enumerator `CFB_DISPLAY_ROWS`

enumerator `CFB_DISPLAY_COLS`

enum `cfb_font_caps`

Values:

enumerator `CFB_FONT_MONO_VPACKED` = *BIT*(0)

enumerator `CFB_FONT_MONO_HPACKED` = *BIT*(1)

enumerator `CFB_FONT_MSB_FIRST` = *BIT*(2)

Functions

int `cfb_print`(const struct *device* *dev, char *str, uint16_t x, uint16_t y)

Print a string into the framebuffer.

Parameters

- `dev` – Pointer to device structure for driver instance
- `str` – String to print
- `x` – Position in X direction of the beginning of the string
- `y` – Position in Y direction of the beginning of the string

Returns 0 on success, negative value otherwise

int `cfb_framebuffer_clear`(const struct *device* *dev, bool clear_display)

Clear framebuffer.

Parameters

- `dev` – Pointer to device structure for driver instance
- `clear_display` – Clear the display as well

Returns 0 on success, negative value otherwise

int `cfb_framebuffer_invert`(const struct *device* *dev)

Invert Pixels.

Parameters

- `dev` – Pointer to device structure for driver instance

Returns 0 on success, negative value otherwise

int `cfb_framebuffer_finalize`(const struct *device* *dev)

Finalize framebuffer and write it to display RAM, invert or reorder pixels if necessary.

Parameters

- `dev` – Pointer to device structure for driver instance

Returns 0 on success, negative value otherwise

int `cfb_get_display_parameter`(const struct *device* *dev, enum *cfb_display_param*)

Get display parameter.

Parameters

- `dev` – Pointer to device structure for driver instance
- `cfb_display_param` – One of the display parameters

Returns Display parameter value

int `cfb_framebuffer_set_font`(const struct *device* *dev, uint8_t idx)

Set font.

Parameters

- `dev` – Pointer to device structure for driver instance
- `idx` – Font index

Returns 0 on success, negative value otherwise

int `cfb_get_font_size`(const struct *device* *dev, uint8_t idx, uint8_t *width, uint8_t *height)

Get font size.

Parameters

- `dev` – Pointer to device structure for driver instance
- `idx` – Font index
- `width` – Pointers to the variable where the font width will be stored.
- `height` – Pointers to the variable where the font height will be stored.

Returns 0 on success, negative value otherwise

int `cfb_get_numof_fonts`(const struct *device* *dev)

Get number of fonts.

Parameters

- `dev` – Pointer to device structure for driver instance

Returns number of fonts

int `cfb_framebuffer_init`(const struct *device* *dev)

Initialize Character Framebuffer.

Parameters

- `dev` – Pointer to device structure for driver instance

Returns 0 on success, negative value otherwise

struct `cfb_font`

#include <cfb.h>

7.9 Error Detection And Correction (EDAC) API

7.9.1 API Reference

group `edac`

Enums

enum `edac_error_type`

EDAC error type.

Values:

enumerator `EDAC_ERROR_TYPE_DRAM_COR` = *BIT*(0)

Correctable error type

enumerator `EDAC_ERROR_TYPE_DRAM_UC` = *BIT*(1)

Uncorrectable error type

Functions

static inline int `edac_inject_set_param1`(const struct *device* *dev, uint64_t value)

Set injection parameter param1.

Set first error injection parameter value.

Parameters

- `dev` – Pointer to the device structure
- `value` – First injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, other error code otherwise

static inline int `edac_inject_get_param1`(const struct *device* *dev, uint64_t *value)

Get injection parameter param1.

Get first error injection parameter value.

Parameters

- `dev` – Pointer to the device structure
- `value` – Pointer to the first injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int `edac_inject_set_param2`(const struct *device* *dev, uint64_t value)

Set injection parameter param2.

Set second error injection parameter value.

Parameters

- `dev` – Pointer to the device structure
- `value` – Second injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int `edac_inject_get_param2`(const struct *device* *dev, uint64_t *value)

Get injection parameter param2.

Parameters

- `dev` – Pointer to the device structure
- `value` – Pointer to the second injection parameter

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int `edac_inject_set_error_type`(const struct *device* *dev, uint32_t error_type)

Set error type value.

Set the value of error type to be injected

Parameters

- `dev` – Pointer to the device structure
- `error_type` – Error type value

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int `edac_inject_get_error_type`(const struct *device* *dev, uint32_t *error_type)

Get error type value.

Get the value of error type to be injected

Parameters

- `dev` – Pointer to the device structure
- `error_type` – Pointer to error type value

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int `edac_inject_error_trigger`(const struct *device* *dev)

Set injection control.

Trigger error injection.

Parameters

- `dev` – Pointer to the device structure

Return values

- `-ENOSYS` – if the optional interface is not implemented
- `0` – on success, error code otherwise

static inline int `edac_ecc_error_log_get`(const struct *device* *dev, uint64_t *value)

Get ECC Error Log.

Read value of ECC Error Log.

Parameters

- `dev` – Pointer to the device structure
- `value` – Pointer to the ECC Error Log value

Return values

- `0` – on success, error code otherwise
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int `edac_ecc_error_log_clear`(const struct *device* *dev)

Clear ECC Error Log.

Clear value of ECC Error Log.

Parameters

- `dev` – Pointer to the device structure

Return values

- `0` – on success, error code otherwise
- `-ENOSYS` – if the mandatory interface is not implemented

static inline int edac_parity_error_log_get(const struct *device* *dev, uint64_t *value)

Get Parity Error Log.

Read value of Parity Error Log.

Parameters

- dev – Pointer to the device structure
- value – Pointer to the parity Error Log value

Return values

- 0 – on success, error code otherwise
- -ENOSYS – if the mandatory interface is not implemented

static inline int edac_parity_error_log_clear(const struct *device* *dev)

Clear Parity Error Log.

Clear value of Parity Error Log.

Parameters

- dev – Pointer to the device structure

Return values

- 0 – on success, error code otherwise
- -ENOSYS – if the mandatory interface is not implemented

static inline int edac_errors_cor_get(const struct *device* *dev)

Get number of correctable errors.

Parameters

- dev – Pointer to the device structure

Return values

- num – Number of correctable errors
- -ENOSYS – if the mandatory interface is not implemented

static inline int edac_errors_uc_get(const struct *device* *dev)

Get number of uncorrectable errors.

Parameters

- dev – Pointer to the device structure

Return values

- num – Number of uncorrectable errors
- -ENOSYS – if the mandatory interface is not implemented

static inline int edac_notify_callback_set(const struct *device* *dev, edac_notify_callback_f cb)

Register callback function for memory error exception

This callback runs in interrupt context

Parameters

- dev – EDAC driver device to install callback
- cb – Callback function pointer

Return values

- 0 – on success, error code otherwise
- -ENOSYS – if the mandatory interface is not implemented

struct edac_driver_api

#include <edac.h> EDAC driver API.

This is the mandatory API any EDAC driver needs to expose.

7.10 File Systems

Zephyr RTOS Virtual Filesystem Switch (VFS) allows applications to mount multiple file systems at different mount points (e.g., `/fatfs` and `/lfs`). The mount point data structure contains all the necessary information required to instantiate, mount, and operate on a file system. The File system Switch decouples the applications from directly accessing an individual file system's specific API or internal functions by introducing file system registration mechanisms.

In Zephyr, any file system implementation or library can be plugged into or pulled out through a file system registration API. Each file system implementation must have a globally unique integer identifier; use *FS_TYPE_EXTERNAL_BASE* to avoid clashes with in-tree identifiers.

```
int fs_register(int type, const struct fs_file_system_t *fs);

int fs_unregister(int type, const struct fs_file_system_t *fs);
```

Zephyr RTOS supports multiple instances of a file system by making use of the mount point as the disk volume name, which is used by the file system library while formatting or mounting a disk.

A file system is declared as:

```
static struct fs_mount_t mp = {
.type = FS_FATFS,
.mnt_point = FATFS_MNTP,
.fs_data = &fat_fs,
};
```

where

- `FS_FATFS` is the file system type like FATFS or LittleFS.
- `FATFS_MNTP` is the mount point where the file system will be mounted.
- `fat_fs` is the file system data which will be used by fs_mount() API.

7.10.1 Samples

Samples for the VFS are mainly supplied in `samples/subsys/fs`, although various examples of the VFS usage are provided as important functionalities in samples for different subsystems. Here is the list of samples worth looking at:

- `samples/subsys/fs/fat_fs` is an example of FAT file system usage with SDHC media;
- `samples/subsys/shell/fs` **is an example of Shell fs subsystem, using internal flash partition** formatted to LittleFS;
- `samples/subsys/usb/mass/` **example of USB Mass Storage device that uses FAT FS driver with RAM** or SPI connected FLASH, or LittleFS in flash, depending on the sample configuration.

7.10.2 API Reference

group `file_system_api`

File System APIs.

fs_open open and creation mode flags

FS_O_READ
: Open for read flag

FS_O_WRITE
: Open for write flag

FS_O_RDWR
: Open for read-write flag combination

FS_O_MODE_MASK
: Bitmask for read and write flags

FS_O_CREATE
: Create file if it does not exist

FS_O_APPEND
: Open/create file for append

FS_O_FLAGS_MASK
: Bitmask for open/create flags

FS_O_MASK
: Bitmask for open flags

fs_seek whence parameter values

FS_SEEK_SET
: Seek from the beginning of file

FS_SEEK_CUR
: Seek from a current position

FS_SEEK_END
: Seek from the end of file

Defines

FS_MOUNT_FLAG_NO_FORMAT
: Flag prevents formatting device if requested file system not found

FS_MOUNT_FLAG_READ_ONLY
: Flag makes mounted file system read-only

FS_MOUNT_FLAG_AUTOMOUNT

Flag used in pre-defined mount structures that are to be mounted on startup.

This flag has no impact in user-defined mount structures.

FSTAB_ENTRY_DT_MOUNT_FLAGS(node_id)

FS_FSTAB_ENTRY(node_id)

The name under which a zephyr,fstab entry mount structure is defined.

FS_FSTAB_DECLARE_ENTRY(node_id)

Generate a declaration for the externally defined fstab entry.

This will evaluate to the name of a struct *fs_mount_t* object.

Enums

enum fs_dir_entry_type

Values:

enumerator FS_DIR_ENTRY_FILE = 0

Identifier for file entry

enumerator FS_DIR_ENTRY_DIR

Identifier for directory entry

enum [anonymous]

Enumeration to uniquely identify file system types.

Zephyr supports in-tree file systems and external ones. Each requires a unique identifier used to register the file system implementation and to associate a mount point with the file system type. This anonymous enum defines global identifiers for the in-tree file systems.

External file systems should be registered using unique identifiers starting at FS_TYPE_EXTERNAL_BASE. It is the responsibility of applications that use external file systems to ensure that these identifiers are unique if multiple file system implementations are used by the application.

Values:

enumerator FS_FATFS = 0

Identifier for in-tree FatFS file system.

enumerator FS_LITTLEFS

Identifier for in-tree LittleFS file system.

enumerator FS_TYPE_EXTERNAL_BASE

Base identifier for external file systems.

Functions

static inline void `fs_file_t_init`(struct *fs_file_t* *zfp)

Initialize *fs_file_t* object.

Initializes the *fs_file_t* object; the function needs to be invoked on object before first use with fs_open.

Parameters

- `zfp` – Pointer to file object

static inline void `fs_dir_t_init`(struct *fs_dir_t* *zdp)

Initialize *fs_dir_t* object.

Initializes the *fs_dir_t* object; the function needs to be invoked on object before first use with fs_opendir.

Parameters

- `zdp` – Pointer to file object

int `fs_open`(struct *fs_file_t* *zfp, const char *file_name, fs_mode_t flags)

Open or create file.

Opens or possibly creates a file and associates a stream with it.

`flags` can be 0 or a binary combination of one or more of the following identifiers:

- `FS_O_READ` open for read
- `FS_O_WRITE` open for write
- `FS_O_RDWR` open for read/write (`FS_O_READ | FS_O_WRITE`)
- `FS_O_CREATE` create file if it does not exist
- `FS_O_APPEND` move to end of file before each write

If `flags` are set to 0 the function will attempt to open an existing file with no read/write access; this may be used to e.g. check if the file exists.

Parameters

- `zfp` – Pointer to a file object
- `file_name` – The name of a file to open
- `flags` – The mode flags

Return values

- `0` – on success;
- `-EINVAL` – when a bad file name is given;
- `-EROFS` – when opening read-only file for write, or attempting to create a file on a system that has been mounted with the FS_MOUNT_FLAG_READ_ONLY flag;
- `-ENOENT` – when the file path is not possible (bad mount point);
- `<0` – an other negative errno code, depending on a file system back-end.

int `fs_close`(struct *fs_file_t* *zfp)

Close file.

Flushes the associated stream and closes the file.

Parameters

- `zfp` – Pointer to the file object

Return values

- `0` – on success;
- `<0` – a negative errno code on error.

int `fs_unlink`(const char *path)

Unlink file.

Deletes the specified file or directory

Parameters

- `path` – Path to the file or directory to delete

Return values

- `0` – on success;
- `-EROFS` – if file is read-only, or when file system has been mounted with the FS_MOUNT_FLAG_READ_ONLY flag;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_rename`(const char *from, const char *to)

Rename file or directory.

Performs a rename and / or move of the specified source path to the specified destination. The source path can refer to either a file or a directory. All intermediate directories in the destination path must already exist. If the source path refers to a file, the destination path must contain a full filename path, rather than just the new parent directory. If an object already exists at the specified destination path, this function causes it to be unlinked prior to the rename (i.e., the destination gets clobbered).

Note: Current implementation does not allow moving files between mount points.

Parameters

- `from` – The source path
- `to` – The destination path

Return values

- `0` – on success;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error.

ssize_t `fs_read`(struct *fs_file_t* *zfp, void *ptr, size_t size)

Read file.

Reads up to `size` bytes of data to `ptr` pointed buffer, returns number of bytes read. A returned value may be lower than `size` if there were fewer bytes available than requested.

Parameters

- `zfp` – Pointer to the file object
- `ptr` – Pointer to the data buffer
- `size` – Number of bytes to be read

Return values

- `>=0` – a number of bytes read, on success;
- `<0` – a negative errno code on error.

ssize_t `fs_write`(struct *fs_file_t* *zfp, const void *ptr, size_t size)

Write file.

Attempts to write `size` number of bytes to the specified file. If a negative value is returned from the function, the file pointer has not been advanced. If the function returns a non-negative number that is lower than `size`, the global `errno` variable should be checked for an error code, as the device may have no free space for data.

Parameters

- `zfp` – Pointer to the file object
- `ptr` – Pointer to the data buffer
- `size` – Number of bytes to be written

Return values

- `>=0` – a number of bytes written, on success;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_seek`(struct *fs_file_t* *zfp, off_t offset, int whence)

Seek file.

Moves the file position to a new location in the file. The `offset` is added to file position based on the `whence` parameter.

Parameters

- `zfp` – Pointer to the file object
- `offset` – Relative location to move the file pointer to
- `whence` – Relative location from where offset is to be calculated.
  - `FS_SEEK_SET` for the beginning of the file;
  - `FS_SEEK_CUR` for the current position;
  - `FS_SEEK_END` for the end of the file.

Return values

- `0` – on success;
- `-ENOTSUP` – if not supported by underlying file system driver;
- `<0` – an other negative errno code on error.

off_t `fs_tell`(struct *fs_file_t* *zfp)

Get current file position.

Retrieves and returns the current position in the file stream.

The current revision does not validate the file object.

Parameters

- `zfp` – Pointer to the file object

Return values

- `>=` – 0 a current position in file;
- `-ENOTSUP` – if not supported by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_truncate`(struct *fs_file_t* *zfp, off_t length)

Truncate or extend an open file to a given size.

Truncates the file to the new length if it is shorter than the current size of the file. Expands the file if the new length is greater than the current size of the file. The expanded region would be filled with zeroes.

Note: In the case of expansion, if the volume got full during the expansion process, the function will expand to the maximum possible length and return success. Caller should check if the expanded size matches the requested length.

Parameters

- `zfp` – Pointer to the file object
- `length` – New size of the file in bytes

Return values

- `0` – on success;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_sync`(struct *fs_file_t* *zfp)

Flush cached write data buffers of an open file.

The function flushes the cache of an open file; it can be invoked to ensure data gets written to the storage media immediately, e.g. to avoid data loss in case if power is removed unexpectedly.

Note: Closing a file will cause caches to be flushed correctly so the function need not be called when the file is being closed.

Parameters

- `zfp` – Pointer to the file object

Return values

- `0` – on success;
- `<0` – a negative errno code on error.

int `fs_mkdir`(const char *path)

Directory create.

Creates a new directory using specified path.

Parameters

- `path` – Path to the directory to create

Return values

- `0` – on success;
- `-ENOTSUP` – when not implemented by underlying file system driver;
- `<0` – an other negative errno code on error

int `fs_opendir`(struct *fs_dir_t* *zdp, const char *path)

Directory open.

Opens an existing directory specified by the path.

Parameters

- `zdp` – Pointer to the directory object
- `path` – Path to the directory to open

Return values

- `0` – on success;
- `<0` – a negative errno code on error.

int `fs_readdir`(struct *fs_dir_t* *zdp, struct *fs_dirent* *entry)

Directory read entry.

Reads directory entries of an open directory. In end-of-dir condition, the function will return 0 and set the `entry->name[0]` to 0.

Note: : Most existing underlying file systems do not generate POSIX special directory entries "." or "..". For consistency the abstraction layer will remove these from lower layer results so higher layers see consistent results.

Parameters

- `zdp` – Pointer to the directory object
- `entry` – Pointer to zfs_dirent structure to read the entry into

Return values

- `0` – on success or end-of-dir;;
- `<0` – a negative errno code on error.

int `fs_closedir`(struct *fs_dir_t* *zdp)

Directory close.

Closes an open directory.

Parameters

- `zdp` – Pointer to the directory object

Return values

- `0` – on success;
- `<0` – a negative errno code on error.

int `fs_mount`(struct *fs_mount_t* *mp)

Mount filesystem.

Perform steps needed for mounting a file system like calling the file system specific mount function and adding the mount point to mounted file system list.

Note: Current implementation of ELM FAT driver allows only following mount points: "/RAM:","/NAND:","/CF:","/SD:","/SD2:","/USB:","/USB2:","/USB3:" or mount points that consist of single digit, e.g: "/0:", "/1:" and so forth.

Parameters

- `mp` – Pointer to the *fs_mount_t* structure. Referenced object is not changed if the mount operation failed. A reference is captured in the fs infrastructure if the mount operation succeeds, and the application must not mutate the structure contents until fs_unmount is successfully invoked on the same pointer.

Return values

- `0` – on success;
- `-ENOENT` – when file system type has not been registered;
- `-ENOTSUP` – when not supported by underlying file system driver;
- `-EROFS` – if system requires formatting but `FS_MOUNT_FLAG_READ_ONLY` has been set;
- `<0` – an other negative errno code on error.

int `fs_unmount`(struct *fs_mount_t* *mp)

Unmount filesystem.

Perform steps needed to unmount a file system like calling the file system specific unmount function and removing the mount point from mounted file system list.

Parameters

- `mp` – Pointer to the *fs_mount_t* structure

Return values

- `0` – on success;
- `-EINVAL` – if no system has been mounted at given mount point;
- `-ENOTSUP` – when not supported by underlying file system driver;
- `<0` – an other negative errno code on error.

int `fs_readmount`(int *index, const char **name)

Get path of mount point at index.

This function iterates through the list of mount points and returns the directory name of the mount point at the given `index`. On success `index` is incremented and `name` is set to the mount directory name. If a mount point with the given `index` does not exist, `name` will be set to `NULL`.

Parameters

- `index` – Pointer to mount point index
- `name` – Pointer to pointer to path name

Return values

- `0` – on success;
- `-ENOENT` – if there is no mount point with given index.

int `fs_stat`(const char *path, struct *fs_dirent* *entry)

File or directory status.

Checks the status of a file or directory specified by the `path`.

Note: The file on a storage device may not be updated until it is closed.

Parameters

- `path` – Path to the file or directory
- `entry` – Pointer to the zfs_dirent structure to fill if the file or directory exists.

Return values

- 0 – on success;
- <0 – negative errno code on error.

int fs_statvfs(const char *path, struct *fs_statvfs* *stat)

Retrieves statistics of the file system volume.

Returns the total and available space in the file system volume.

Parameters

- path – Path to the mounted directory
- stat – Pointer to the zfs_statvfs structure to receive the fs statistics

Return values

- 0 – on success;
- -ENOTSUP – when not implemented by underlying file system driver;
- <0 – an other negative errno code on error.

int fs_register(int type, const struct *fs_file_system_t* *fs)

Register a file system.

Register file system with virtual file system.

Parameters

- type – Type of file system (ex: FS_FATFS)
- fs – Pointer to File system

Return values

- 0 – on success;
- <0 – negative errno code on error.

int fs_unregister(int type, const struct *fs_file_system_t* *fs)

Unregister a file system.

Unregister file system from virtual file system.

Parameters

- type – Type of file system (ex: FS_FATFS)
- fs – Pointer to File system

Return values

- 0 – on success;
- <0 – negative errno code on error.

struct fs_mount_t

#include <fs.h> File system mount info structure.

Param node Entry for the fs_mount_list list

Param type File system type

Param mnt_point Mount point directory name (ex: "/fatfs")

Param fs_data Pointer to file system specific data

Param storage_dev Pointer to backend storage device

Param mountp_len Length of Mount point string

Param fs Pointer to File system interface of the mount point

Param flags Mount flags

struct `fs_dirent`

#include <fs.h> Structure to receive file or directory information.

Used in functions that reads the directory entries to get file or directory information.

Param dir_entry_type Whether file or directory

- FS_DIR_ENTRY_FILE
- FS_DIR_ENTRY_DIR

Param name Name of directory or file

Param size Size of file. 0 if directory

struct `fs_statvfs`

#include <fs.h> Structure to receive volume statistics.

Used to retrieve information about total and available space in the volume.

Param f_bsize Optimal transfer block size

Param f_frsize Allocation unit size

Param f_blocks Size of FS in f_frsize units

Param f_bfree Number of free blocks

struct `fs_file_t`

#include <fs_interface.h> File object representing an open file.

The object needs to be initialized with function *fs_file_t_init()*.

Param Pointer to FATFS file object structure

Param mp Pointer to mount point structure

struct `fs_dir_t`

#include <fs_interface.h> Directory object representing an open directory.

The object needs to be initialized with function *fs_dir_t_init()*.

Param dirp Pointer to directory object structure

Param mp Pointer to mount point structure

struct `fs_file_system_t`

#include <fs_sys.h> File System interface structure.

Param open Opens or creates a file, depending on flags given

Param read Reads nbytes number of bytes

Param write Writes nbytes number of bytes

Param lseek Moves the file position to a new location in the file

Param tell Retrieves the current position in the file

Param truncate Truncates/expands the file to the new length

Param sync Flushes the cache of an open file

Param close Flushes the associated stream and closes the file

Param opendir Opens an existing directory specified by the path

Param readdir Reads directory entries of an open directory

Param closedir Closes an open directory

Param mount Mounts a file system

Param unmount Unmounts a file system

Param unlink Deletes the specified file or directory

Param rename Renames a file or directory

Param mkdir Creates a new directory using specified path

Param stat Checks the status of a file or directory specified by the path

Param statvfs Returns the total and available space on the file system volume

7.11 Iterable Sections

This page contains the reference documentation for the iterable sections APIs, which can be used for defining iterable areas of equally-sized data structures, that can be iterated on using *STRUCT_SECTION_FOREACH()*.

7.11.1 Usage

Iterable section elements are typically used by defining the data structure and associated initializer in a common header file, so that they can be instantiated anywhere in the code base.

```
struct my_data {
        int a, b;
};

#define DEFINE_DATA(name, _a, _b) \
        STRUCT_SECTION_ITERABLE(my_data, name) = { \
                .a = _a, \
                .b = _b, \
        }

...

DEFINE_DATA(d1, 1, 2);
DEFINE_DATA(d2, 3, 4);
DEFINE_DATA(d3, 5, 6);
```

Then the linker has to be setup to place the place the structure in a contiguous segment using one of the linker macros such as *ITERABLE_SECTION_RAM()* or *ITERABLE_SECTION_ROM()*. Custom linker snippets are normally declared using one of the `zephyr_linker_sources()` CMake functions, using the appropriate section identifier, `DATA_SECTIONS` for RAM structures and `SECTIONS` for ROM ones.

```
# CMakeLists.txt
zephyr_linker_sources(DATA_SECTIONS iterables.ld)
```

```
# iterables.ld
ITERABLE_SECTION_RAM(my_data, 4)
```

The data can then be accessed using *STRUCT_SECTION_FOREACH()*.

```
STRUCT_SECTION_FOREACH(my_data, data) {
        printk("%p: a: %d, b: %d\n", data, data->a, data->b);
}
```

Note: The linker is going to place the entries sorted by name, so the example above would visit `d1`, `d2` and `d3` in that order, regardless of how they were defined in the code.

7.11.2 API Reference

group `iterable_section_apis`

Iterable Sections APIs.

Defines

`ITERABLE_SECTION_ROM(struct_type, subalign)`

Define a read-only iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-only data.

Note that this keeps the symbols in the image even though they are not being directly referenced. Use this when symbols are indirectly referenced by iterating through the section.

`ITERABLE_SECTION_ROM_GC_ALLOWED(struct_type, subalign)`

Define a garbage collectable read-only iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-only data.

Note that the symbols within the section can be garbage collected.

`ITERABLE_SECTION_RAM(struct_type, subalign)`

Define a read-write iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-write data that is modified at runtime.

Note that this keeps the symbols in the image even though they are not being directly referenced. Use this when symbols are indirectly referenced by iterating through the section.

`ITERABLE_SECTION_RAM_GC_ALLOWED(struct_type, subalign)`

Define a garbage collectable read-write iterable section output.

Define an output section which will set up an iterable area of equally-sized data structures. For use with *STRUCT_SECTION_ITERABLE()*. Input sections will be sorted by name, per ld's SORT_BY_NAME.

This macro should be used for read-write data that is modified at runtime.

Note that the symbols within the section can be garbage collected.

STRUCT_SECTION_ITERABLE(struct_type, name)

Defines a new iterable section.

Convenience helper combining __in_section() and Z_DECL_ALIGN(). The section name is the struct type prepended with an underscore. The subsection is "static" and the subsubsection is the variable name.

In the linker script, create output sections for these using *ITERABLE_SECTION_ROM()* or *ITERABLE_SECTION_RAM()*.

STRUCT_SECTION_ITERABLE_ALTERNATE(out_type, struct_type, name)

Defines an alternate data type iterable section.

Special variant of *STRUCT_SECTION_ITERABLE()*, for placing alternate data types within the iterable section of a specific data type. The data type sizes and semantics must be equivalent!

STRUCT_SECTION_FOREACH(struct_type, iterator)

Iterate over a specified iterable section.

Iterator for structure instances gathered by *STRUCT_SECTION_ITERABLE()*. The linker must provide a _<struct_type>_list_start symbol and a _<struct_type>_list_end symbol to mark the start and the end of the list of struct objects to iterate over. This is normally done using *ITERABLE_SECTION_ROM()* or *ITERABLE_SECTION_RAM()* in the linker script.

7.12 Formatted Output

Applications as well as Zephyr itself requires infrastructure to format values for user consumption. The standard C99 library `*printf()` functionality fulfills this need for streaming output devices or memory buffers, but in an embedded system devices may not accept streamed data and memory may not be available to store the formatted output.

Internal Zephyr API traditionally provided this both for `printk()` and for Zephyr's internal minimal libc, but with separate internal interfaces. Logging, tracing, shell, and other applications made use of either these APIs or standard libc routines based on build options.

The *cbprintf()* public APIs convert C99 format strings and arguments, providing output produced one character at a time through a callback mechanism, replacing the original internal functions and providing support for almost all C99 format specifications. Existing use of `s*printf()` C libraries in Zephyr can be converted to *snprintfcb()* to avoid pulling in libc implementations.

Several Kconfig options control the set of features that are enabled, allowing some control over features and memory usage:

- `CONFIG_CBPRINTF_FULL_INTEGRAL` or `CONFIG_CBPRINTF_REDUCED_INTEGRAL`
- `CONFIG_CBPRINTF_FP_SUPPORT`
- `CONFIG_CBPRINTF_FP_A_SUPPORT`
- `CONFIG_CBPRINTF_FP_ALWAYS_A`
- `CONFIG_CBPRINTF_N_SPECIFIER`

`CONFIG_CBPRINTF_LIBC_SUBSTS` can be used to provide functions that behave like standard libc functions but use the selected cbprintf formatter rather than pulling in another formatter from libc.

In addition `CONFIG_CBPRINTF_NANO` can be used to revert back to the very space-optimized but limited formatter used for `printk()` before this capability was added.

7.12.1 Cbprintf Packaging

Typically, strings are formatted synchronously when a function from `printf` family is called. However, there are cases when it is beneficial that formatting is deferred. In that case, a state (format string

and arguments) must be captured. Such state forms a self-contained package which contains format string and arguments. Additionally, package contains copies of all strings which are part of a format string (format string or any %s argument) and are identifed as the one located in the read write memory. Package primary content resembles va_list stack frame thus standard formatting functions are used to process a package. Since package contains data which is processed as va_list frame, strict alignment must be maintained. Due to required padding, size of the package depends on alignment. When package is copied, it should be copied to a memory block with the same alignment as origin.

Package can be created using two methods:

- runtime - using *cbprintf_package()* or *cbvprintf_package()*. This method scans format string and based on detected format specifiers builds the package.
- static - types of arguments are detected at compile time by the preprocessor and package is created as simple assignments to a provided memory. This method is significantly faster than runtime (more than 15 times) but has following limitations: requires `_Generic` keyword (C11 feature) to be supported by the compiler and can only create a package that is known to have no string arguments (%s). *CBPRINTF_MUST_RUNTIME_PACKAGE* can be used to determine at compile time if static packaging can be applied. Macro determines need for runtime packaging based on presence of char pointers in the argument list so there are cases when it will be false positive, e.g. %p with char pointer.

Several Kconfig options control behavior of the packaging:

- `CONFIG_CBPRINTF_PACKAGE_LONGDOUBLE`
- `CONFIG_CBPRINTF_STATIC_PACKAGE_CHECK_ALIGNMENT`

Cbprintf package format

Format of the package contains paddings which are platform specific. Package consists of header which contains size of package (excluding appended strings) and number of appended strings. It is followed by the arguments which contains alignment paddings and resembles *va_list* stack frame. Finally, package optionally contains appended strings. Each string contains 1 byte header which contains index of the location where address argument is stored. During packaging address is set to null and before string formatting it is updated to point to the current string location within the package. Updating address argument must happen just before string formatting since address changes whenever package is copied.

Header sizeof(void *)	1 byte: Argument list size including header and *fmt* (in 32 bit words)
	1 byte: Number of appended strings
	platform specific padding to sizeof(void *)
Arguments	Pointer to *fmt* (or null if *fmt* is appended to the package)
	(optional padding for platform specific alignment)
	argument 0
	(optional padding for platform specific alignment)
	argument 1
	...
Appended strings	1 byte: Index within the package to the location of associated argument
	Null terminated string
	...

Warning: If CONFIG_MINIMAL_LIBC is selected in combination with CONFIG_CBPRINTF_NANO formatting with C standard library functions like printf or snprintf is limited. Among other things the %n specifier, most format flags, precision control, and floating point are not supported.

7.12.2 API Reference

group cbprintf_apis

Defines

CBPRINTF_PACKAGE_ALIGNMENT

Required alignment of the buffer used for packaging.

CBPRINTF_MUST_RUNTIME_PACKAGE(skip, ...)

Determine if string must be packaged in run time.

Static packaging can be applied if size of the package can be determined at compile time. In general, package size can be determined at compile time if there are no string arguments which might be copied into package body if they are considered transient.

Parameters

- skip – number of read only string arguments in the parameter list. It shall be non-zero if there are known read only string arguments present in the string (e.g. function name prefix in the log message).
- ... – String with arguments.

Return values

- 1 – if string must be packaged in run time.
- 0 – string can be statically packaged.

CBPRINTF_STATIC_PACKAGE(packaged, inlen, outlen, align_offset, flags, ...)

Statically package string.

Build string package from formatted string. It assumes that formatted string is in the read only memory.

If _Generic is not supported then runtime packaging is performed.

Parameters

- packaged – pointer to where the packaged data can be stored. Pass a null pointer to skip packaging but still calculate the total space required. The data stored here is relocatable, that is it can be moved to another contiguous block of memory. It must be aligned to the size of the longest argument. It is recommended to use CBPRINTF_PACKAGE_ALIGNMENT for alignment.
- inlen – set to the number of bytes available at packaged. If packaged is NULL the value is ignored.
- outlen – variable updated to the number of bytes required to completely store the packed information. If input buffer was too small it is set to -ENOSPC.
- align_offset – input buffer alignment offset in bytes. Where offset 0 means that buffer is aligned to CBPRINTF_PACKAGE_ALIGNMENT. Xtensa requires that packaged is aligned to CBPRINTF_PACKAGE_ALIGNMENT so it must be multiply of CBPRINTF_PACKAGE_ALIGNMENT or 0.

- `flags` – option flags. See Package flags..
- `...` – formatted string with arguments. Format string must be constant.

Typedefs

typedef int (*`cbprintf_cb`)()

Signature for a cbprintf callback function.

This function expects two parameters:

- `c` a character to output. The output behavior should be as if this was cast to an unsigned char.
- `ctx` a pointer to an object that provides context for the output operation.

The declaration does not specify the parameter types. This allows a function like `fputc` to be used without requiring all context pointers to be to a `FILE` object.

Return the value of `c` cast to an unsigned char then back to int, or a negative error code that will be returned from *cbprintf()*.

Functions

int `cbprintf_package`(void *packaged, size_t len, uint32_t flags, const char *format, ...)

Capture state required to output formatted data later.

Like *cbprintf()* but instead of processing the arguments and emitting the formatted results immediately all arguments are captured so this can be done in a different context, e.g. when the output function can block.

In addition to the values extracted from arguments this will ensure that copies are made of the necessary portions of any string parameters that are not confirmed to be stored in read-only memory (hence assumed to be safe to refer to directly later).

Parameters

- `packaged` – pointer to where the packaged data can be stored. Pass a null pointer to store nothing but still calculate the total space required. The data stored here is relocatable, that is it can be moved to another contiguous block of memory. However, under condition that alignment is maintained. It must be aligned to at least the size of a pointer.
- `len` – this must be set to the number of bytes available at `packaged` if it is not null. If `packaged` is null then it indicates hypothetical buffer alignment offset in bytes compared to CBPRINTF_PACKAGE_ALIGNMENT alignment. Buffer alignment offset impacts returned size of the package. Xtensa requires that buffer is always aligned to CBPRINTF_PACKAGE_ALIGNMENT so it must be multiply of CBPRINTF_PACKAGE_ALIGNMENT or 0 when `packaged` is null.
- `flags` – option flags. See Package flags..
- `format` – a standard ISO C format string with characters and conversion specifications.
- `...` – arguments corresponding to the conversion specifications found within `format`.

Return values

- `nonegative` – the number of bytes successfully stored at `packaged`. This will not exceed `len`.

- -EINVAL – if format is not acceptable
- -EFAULT – if packaged alignment is not acceptable
- -ENOSPC – if packaged was not null and the space required to store exceed len.

int cbvprintf_package(void *packaged, size_t len, uint32_t flags, const char *format, va_list ap)

Capture state required to output formatted data later.

Like *cbprintf()* but instead of processing the arguments and emitting the formatted results immediately all arguments are captured so this can be done in a different context, e.g. when the output function can block.

In addition to the values extracted from arguments this will ensure that copies are made of the necessary portions of any string parameters that are not confirmed to be stored in read-only memory (hence assumed to be safe to refer to directly later).

Parameters

- packaged – pointer to where the packaged data can be stored. Pass a null pointer to store nothing but still calculate the total space required. The data stored here is relocatable, that is it can be moved to another contiguous block of memory. The pointer must be aligned to a multiple of the largest element in the argument list.
- len – this must be set to the number of bytes available at packaged. Ignored if packaged is NULL.
- flags – option flags. See Package flags..
- format – a standard ISO C format string with characters and conversion specifications.
- ap – captured stack arguments corresponding to the conversion specifications found within format.

Return values

- nonegative – the number of bytes successfully stored at packaged. This will not exceed len.
- -EINVAL – if format is not acceptable
- -ENOSPC – if packaged was not null and the space required to store exceed len.

int cbprintf_fsc_package(void *in_packaged, size_t in_len, void *packaged, size_t len)

Convert package to fully self-contained (fsc) package.

By default, package does not contain read only strings. However, if needed it may be converted to a fully self-contained package which contains all strings. In order to allow such conversion, original package must be created with CBPRINTF_PACKAGE_ADD_STRING_IDXS flag. Such package will contain necessary data to find read only strings in the package and copy them into package body.

Parameters

- in_packaged – pointer to original package created with CBPRINTF_PACKAGE_ADD_STRING_IDXS.
- in_len – in_packaged length.
- packaged – pointer to location where fully self-contained version of the input package will be written. Pass a null pointer to calculate space required.
- len – must be set to the number of bytes available at packaged. Not used if packaged is null.

Return values

- `nonegative` – the number of bytes successfully stored at `packaged`. This will not exceed `len`. If `packaged` is null, calculated length.
- `-ENOSPC` – if `packaged` was not null and the space required to store exceed `len`.
- `-EINVAL` – if `in_packaged` is null.

int `cbpprintf`(*cbprintf_cb* out, void *ctx, void *packaged)

Generate the output for a previously captured format operation.

Note: Memory indicated by `packaged` will be modified in a non-destructive way, meaning that it could still be reused with this function again.

Parameters

- `out` – the function used to emit each generated character.
- `ctx` – context provided when invoking out
- `packaged` – the data required to generate the formatted output, as captured by *cbprintf_package()* or *cbvprintf_package()*. The alignment requirement on this data is the same as when it was initially created.

Returns the number of characters printed, or a negative error value returned from invoking `out`.

int `cbprintf`(*cbprintf_cb* out, void *ctx, const char *format, ...)

*printf-like output through a callback.

This is essentially printf() except the output is generated character-by-character using the provided `out` function. This allows formatting text of unbounded length without incurring the cost of a temporary buffer.

All formatting specifiers of C99 are recognized, and most are supported if the functionality is enabled.

Note: The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- `out` – the function used to emit each generated character.
- `ctx` – context provided when invoking out
- `format` – a standard ISO C format string with characters and conversion specifications.
- `...` – arguments corresponding to the conversion specifications found within `format`.

Returns the number of characters printed, or a negative error value returned from invoking `out`.

int `cbvprintf`(*cbprintf_cb* out, void *ctx, const char *format, va_list ap)

varargs-aware *printf-like output through a callback.

This is essentially vsprintf() except the output is generated character-by-character using the provided `out` function. This allows formatting text of unbounded length without incurring the cost of a temporary buffer.

Note: This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

Note: The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- out – the function used to emit each generated character.
- ctx – context provided when invoking out
- format – a standard ISO C format string with characters and conversion specifications.
- ap – a reference to the values to be converted.

Returns the number of characters generated, or a negative error value returned from invoking out.

int fprintfcb(FILE *stream, const char *format, ...)

fprintf using Zephyrs cbprintf infrastructure.

return The number of characters printed.

Note: This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

Note: The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- stream – the stream to which the output should be written.
- format – a standard ISO C format string with characters and conversion specifications.
- ... – arguments corresponding to the conversion specifications found within format.

int vfprintfcb(FILE *stream, const char *format, va_list ap)

vfprintf using Zephyrs cbprintf infrastructure.

Note: This function is available only when CONFIG_CBPRINTF_LIBC_SUBSTS is selected.

Note: The functionality of this function is significantly reduced when CONFIG_CBPRINTF_NANO is selected.

Parameters

- stream – the stream to which the output should be written.

- `format` – a standard ISO C format string with characters and conversion specifications.
- `ap` – a reference to the values to be converted.

Returns The number of characters printed.

int `printfcb`(const char *format, ...)

printf using Zephyrs cbprintf infrastructure.

Note: This function is available only when `CONFIG_CBPRINTF_LIBC_SUBSTS` is selected.

Note: The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- `format` – a standard ISO C format string with characters and conversion specifications.
- `...` – arguments corresponding to the conversion specifications found within `format`.

Returns The number of characters printed.

int `vprintfcb`(const char *format, va_list ap)

vprintf using Zephyrs cbprintf infrastructure.

Note: This function is available only when `CONFIG_CBPRINTF_LIBC_SUBSTS` is selected.

Note: The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- `format` – a standard ISO C format string with characters and conversion specifications.
- `ap` – a reference to the values to be converted.

Returns The number of characters printed.

int `snprintfcb`(char *str, size_t size, const char *format, ...)

snprintf using Zephyrs cbprintf infrastructure.

Note: This function is available only when `CONFIG_CBPRINTF_LIBC_SUBSTS` is selected.

Note: The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- `str` – where the formatted content should be written

- `size` – maximum number of chaacters for the formatted output, including the terminating null byte.
- `format` – a standard ISO C format string with characters and conversion specifications.
- `...` – arguments corresponding to the conversion specifications found within `format`.

Returns The number of characters that would have been written to `str`, excluding the terminating null byte. This is greater than the number actually written if `size` is too small.

int `vsnprintfcb`(char *str, size_t size, const char *format, va_list ap)

vsnprintf using Zephyrs cbprintf infrastructure.

Note: This function is available only when `CONFIG_CBPRINTF_LIBC_SUBSTS` is selected.

Note: The functionality of this function is significantly reduced when `CONFIG_CBPRINTF_NANO` is selected.

Parameters

- `str` – where the formatted content should be written
- `size` – maximum number of chaacters for the formatted output, including the terminating null byte.
- `format` – a standard ISO C format string with characters and conversion specifications.
- `ap` – a reference to the values to be converted.

Returns The number of characters that would have been written to `str`, excluding the terminating null byte. This is greater than the number actually written if `size` is too small.

7.13 Kernel Services

The Zephyr kernel lies at the heart of every Zephyr application. It provides a low footprint, high performance, multi-threaded execution environment with a rich set of available features. The rest of the Zephyr ecosystem, including device drivers, networking stack, and application-specific code, uses the kernel's features to create a complete application.

The configurable nature of the kernel allows you to incorporate only those features needed by your application, making it ideal for systems with limited amounts of memory (as little as 2 KB!) or with simple multi-threading requirements (such as a set of interrupt handlers and a single background task). Examples of such systems include: embedded sensor hubs, environmental sensors, simple LED wearable, and store inventory tags.

Applications requiring more memory (50 to 900 KB), multiple communication devices (like Wi-Fi and Bluetooth Low Energy), and complex multi-threading, can also be developed using the Zephyr kernel. Examples of such systems include: fitness wearables, smart watches, and IoT wireless gateways.

7.13.1 Scheduling, Interrupts, and Synchronization

These pages cover basic kernel services related to thread scheduling and synchronization.

Threads

Note: There is also limited support for using *Zephyr Without Threads*.

This section describes kernel services for creating, scheduling, and deleting independently executable threads of instructions.

A *thread* is a kernel object that is used for application processing that is too lengthy or too complex to be performed by an ISR.

Any number of threads can be defined by an application (limited only by available RAM). Each thread is referenced by a *thread id* that is assigned when the thread is spawned.

A thread has the following key properties:

- A **stack area**, which is a region of memory used for the thread's stack. The **size** of the stack area can be tailored to conform to the actual needs of the thread's processing. Special macros exist to create and work with stack memory regions.
- A **thread control block** for private kernel bookkeeping of the thread's metadata. This is an instance of type `k_thread`.
- An **entry point function**, which is invoked when the thread is started. Up to 3 **argument values** can be passed to this function.

- A **scheduling priority**, which instructs the kernel's scheduler how to allocate CPU time to the thread. (See *Scheduling*.)
- A set of **thread options**, which allow the thread to receive special treatment by the kernel under specific circumstances. (See *Thread Options*.)
- A **start delay**, which specifies how long the kernel should wait before starting the thread.
- An **execution mode**, which can either be supervisor or user mode. By default, threads run in supervisor mode and allow access to privileged CPU instructions, the entire memory address space, and peripherals. User mode threads have a reduced set of privileges. This depends on the `CONFIG_USERSPACE` option. See *User Mode*.

Lifecycle

Thread Creation A thread must be created before it can be used. The kernel initializes the thread control block as well as one end of the stack portion. The remainder of the thread's stack is typically left uninitialized.

Specifying a start delay of *K_NO_WAIT* instructs the kernel to start thread execution immediately. Alternatively, the kernel can be instructed to delay execution of the thread by specifying a timeout value – for example, to allow device hardware used by the thread to become available.

The kernel allows a delayed start to be canceled before the thread begins executing. A cancellation request has no effect if the thread has already started. A thread whose delayed start was successfully canceled must be re-spawned before it can be used.

Thread Termination Once a thread is started it typically executes forever. However, a thread may synchronously end its execution by returning from its entry point function. This is known as **termination**.

A thread that terminates is responsible for releasing any shared resources it may own (such as mutexes and dynamically allocated memory) prior to returning, since the kernel does *not* reclaim them automatically.

In some cases a thread may want to sleep until another thread terminates. This can be accomplished with the *k_thread_join()* API. This will block the calling thread until either the timeout expires, the target thread self-exits, or the target thread aborts (either due to a *k_thread_abort()* call or triggering a fatal error).

Once a thread has terminated, the kernel guarantees that no use will be made of the thread struct. The memory of such a struct can then be re-used for any purpose, including spawning a new thread. Note that the thread must be fully terminated, which presents race conditions where a thread's own logic signals completion which is seen by another thread before the kernel processing is complete. Under normal circumstances, application code should use *k_thread_join()* or *k_thread_abort()* to synchronize on thread termination state and not rely on signaling from within application logic.

Thread Aborting A thread may asynchronously end its execution by **aborting**. The kernel automatically aborts a thread if the thread triggers a fatal error condition, such as dereferencing a null pointer.

A thread can also be aborted by another thread (or by itself) by calling *k_thread_abort()*. However, it is typically preferable to signal a thread to terminate itself gracefully, rather than aborting it.

As with thread termination, the kernel does not reclaim shared resources owned by an aborted thread.

Note: The kernel does not currently make any claims regarding an application's ability to respawn a thread that aborts.

Thread Suspension A thread can be prevented from executing for an indefinite period of time if it becomes **suspended**. The function *k_thread_suspend()* can be used to suspend any thread, including the calling thread. Suspending a thread that is already suspended has no additional effect.

Once suspended, a thread cannot be scheduled until another thread calls *k_thread_resume()* to remove the suspension.

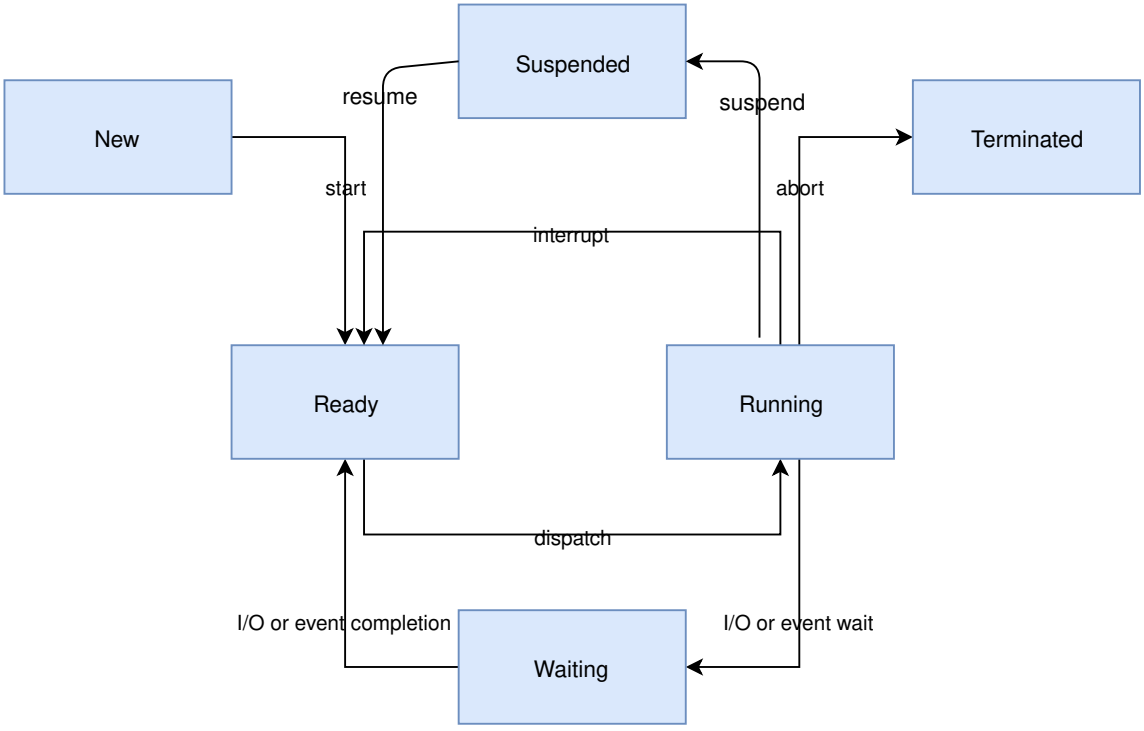
Note: A thread can prevent itself from executing for a specified period of time using *k_sleep()*. However, this is different from suspending a thread since a sleeping thread becomes executable automatically when the time limit is reached.

Thread States A thread that has no factors that prevent its execution is deemed to be **ready**, and is eligible to be selected as the current thread.

A thread that has one or more factors that prevent its execution is deemed to be **unready**, and cannot be selected as the current thread.

The following factors make a thread unready:

- The thread has not been started.
- The thread is waiting for a kernel object to complete an operation. (For example, the thread is taking a semaphore that is unavailable.)
- The thread is waiting for a timeout to occur.
- The thread has been suspended.
- The thread has terminated or aborted.



Thread Stack objects Every thread requires its own stack buffer for the CPU to push context. Depending on configuration, there are several constraints that must be met:

- There may need to be additional memory reserved for memory management structures
- If guard-based stack overflow detection is enabled, a small write- protected memory management region must immediately precede the stack buffer to catch overflows.

- If userspace is enabled, a separate fixed-size privilege elevation stack must be reserved to serve as a private kernel stack for handling system calls.
- If userspace is enabled, the thread's stack buffer must be appropriately sized and aligned such that a memory protection region may be programmed to exactly fit.

The alignment constraints can be quite restrictive, for example some MPUs require their regions to be of some power of two in size, and aligned to its own size.

Because of this, portable code can't simply pass an arbitrary character buffer to *k_thread_create()*. Special macros exist to instantiate stacks, prefixed with K_KERNEL_STACK and K_THREAD_STACK.

Kernel-only Stacks If it is known that a thread will never run in user mode, or the stack is being used for special contexts like handling interrupts, it is best to define stacks using the K_KERNEL_STACK macros.

These stacks save memory because an MPU region will never need to be programmed to cover the stack buffer itself, and the kernel will not need to reserve additional room for the privilege elevation stack, or memory management data structures which only pertain to user mode threads.

Attempts from user mode to use stacks declared in this way will result in a fatal error for the caller.

If CONFIG_USERSPACE is not enabled, the set of K_THREAD_STACK macros have an identical effect to the K_KERNEL_STACK macros.

Thread stacks If it is known that a stack will need to host user threads, or if this cannot be determined, define the stack with K_THREAD_STACK macros. This may use more memory but the stack object is suitable for hosting user threads.

If CONFIG_USERSPACE is not enabled, the set of K_THREAD_STACK macros have an identical effect to the K_KERNEL_STACK macros.

Thread Priorities A thread's priority is an integer value, and can be either negative or non-negative. Numerically lower priorities takes precedence over numerically higher values. For example, the scheduler gives thread A of priority 4 *higher* priority over thread B of priority 7; likewise thread C of priority -2 has higher priority than both thread A and thread B.

The scheduler distinguishes between two classes of threads, based on each thread's priority.

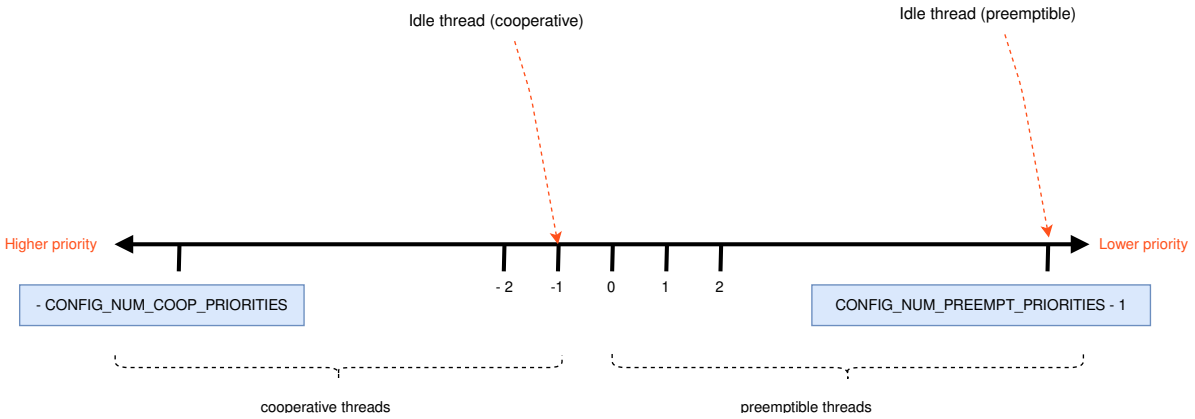
- A *cooperative thread* has a negative priority value. Once it becomes the current thread, a cooperative thread remains the current thread until it performs an action that makes it unready.
- A *preemptible thread* has a non-negative priority value. Once it becomes the current thread, a preemptible thread may be supplanted at any time if a cooperative thread, or a preemptible thread of higher or equal priority, becomes ready.

A thread's initial priority value can be altered up or down after the thread has been started. Thus it is possible for a preemptible thread to become a cooperative thread, and vice versa, by changing its priority.

Note: The scheduler does not make heuristic decisions to re-prioritize threads. Thread priorities are set and changed only at the application's request.

The kernel supports a virtually unlimited number of thread priority levels. The configuration options CONFIG_NUM_COOP_PRIORITIES and CONFIG_NUM_PREEMPT_PRIORITIES specify the number of priority levels for each class of thread, resulting in the following usable priority ranges:

- cooperative threads: (-CONFIG_NUM_COOP_PRIORITIES) to -1
- preemptive threads: 0 to (CONFIG_NUM_PREEMPT_PRIORITIES - 1)



For example, configuring 5 cooperative priorities and 10 preemptive priorities results in the ranges -5 to -1 and 0 to 9, respectively.

Meta-IRQ Priorities When enabled (see `CONFIG_NUM_METAIRQ_PRIORITIES`), there is a special subclass of cooperative priorities at the highest (numerically lowest) end of the priority space: meta-IRQ threads. These are scheduled according to their normal priority, but also have the special ability to preempt all other threads (and other meta-IRQ threads) at lower priorities, even if those threads are cooperative and/or have taken a scheduler lock. Meta-IRQ threads are still threads, however, and can still be interrupted by any hardware interrupt.

This behavior makes the act of unblocking a meta-IRQ thread (by any means, e.g. creating it, calling k_sem_give(), etc.) into the equivalent of a synchronous system call when done by a lower priority thread, or an ARM-like "pended IRQ" when done from true interrupt context. The intent is that this feature will be used to implement interrupt "bottom half" processing and/or "tasklet" features in driver subsystems. The thread, once woken, will be guaranteed to run before the current CPU returns into application code.

Unlike similar features in other OSes, meta-IRQ threads are true threads and run on their own stack (which must be allocated normally), not the per-CPU interrupt stack. Design work to enable the use of the IRQ stack on supported architectures is pending.

Note that because this breaks the promise made to cooperative threads by the Zephyr API (namely that the OS won't schedule other thread until the current thread deliberately blocks), it should be used only with great care from application code. These are not simply very high priority threads and should not be used as such.

Thread Options The kernel supports a small set of *thread options* that allow a thread to receive special treatment under specific circumstances. The set of options associated with a thread are specified when the thread is spawned.

A thread that does not require any thread option has an option value of zero. A thread that requires a thread option specifies it by name, using the `|` character as a separator if multiple options are needed (i.e. combine options using the bitwise OR operator).

The following thread options are supported.

`K_ESSENTIAL` This option tags the thread as an *essential thread*. This instructs the kernel to treat the termination or aborting of the thread as a fatal system error.

By default, the thread is not considered to be an essential thread.

`K_SSE_REGS` This x86-specific option indicate that the thread uses the CPU's SSE registers. Also see *`K_FP_REGS`*.

By default, the kernel does not attempt to save and restore the contents of this register when scheduling the thread.

`K_FP_REGS` This option indicate that the thread uses the CPU's floating point registers. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. (For more information see *Floating Point Services*.)

By default, the kernel does not attempt to save and restore the contents of this register when scheduling the thread.

`K_USER` If `CONFIG_USERSPACE` is enabled, this thread will be created in user mode and will have reduced privileges. See *User Mode*. Otherwise this flag does nothing.

`K_INHERIT_PERMS` If `CONFIG_USERSPACE` is enabled, this thread will inherit all kernel object permissions that the parent thread had, except the parent thread object. See *User Mode*.

Thread Custom Data Every thread has a 32-bit *custom data* area, accessible only by the thread itself, and may be used by the application for any purpose it chooses. The default custom data value for a thread is zero.

Note: Custom data support is not available to ISRs because they operate within a single shared kernel interrupt handling context.

By default, thread custom data support is disabled. The configuration option `CONFIG_THREAD_CUSTOM_DATA` can be used to enable support.

The *k_thread_custom_data_set()* and *k_thread_custom_data_get()* functions are used to write and read a thread's custom data, respectively. A thread can only access its own custom data, and not that of another thread.

The following code uses the custom data feature to record the number of times each thread calls a specific routine.

Note: Obviously, only a single routine can use this technique, since it monopolizes the use of the custom data feature.

```
int call_tracking_routine(void)
{
    uint32_t call_count;

    if (k_is_in_isr()) {
        /* ignore any call made by an ISR */
    } else {
        call_count = (uint32_t)k_thread_custom_data_get();
        call_count++;
        k_thread_custom_data_set((void *)call_count);
    }

    /* do rest of routine's processing */
    ...
}
```

Use thread custom data to allow a routine to access thread-specific information, by using the custom data as a pointer to a data structure owned by the thread.

Implementation

Spawning a Thread A thread is spawned by defining its stack area and its thread control block, and then calling *k_thread_create()*.

The stack area must be defined using *K_THREAD_STACK_DEFINE* or *K_KERNEL_STACK_DEFINE* to ensure it is properly set up in memory.

The size parameter for the stack must be one of three values:

- The original requested stack size passed to K_THREAD_STACK or K_KERNEL_STACK family of stack instantiation macros.
- For a stack object defined with the K_THREAD_STACK family of macros, the return value of *K_THREAD_STACK_SIZEOF()* for that' object.
- For a stack object defined with the K_KERNEL_STACK family of macros, the return value of *K_KERNEL_STACK_SIZEOF()* for that object.

The thread spawning function returns its thread id, which can be used to reference the thread.

The following code spawns a thread that starts immediately.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);
struct k_thread my_thread_data;

k_tid_t my_tid = k_thread_create(&my_thread_data, my_stack_area,
                                 K_THREAD_STACK_SIZEOF(my_stack_area),
                                 my_entry_point,
                                 NULL, NULL, NULL,
                                 MY_PRIORITY, 0, K_NO_WAIT);
```

Alternatively, a thread can be declared at compile time by calling *K_THREAD_DEFINE*. Observe that the macro defines the stack area, control block, and thread id variables automatically.

The following code has the same effect as the code segment above.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_DEFINE(my_tid, MY_STACK_SIZE,
                my_entry_point, NULL, NULL, NULL,
                MY_PRIORITY, 0, 0);
```

Note: The delay parameter to *k_thread_create()* is a *k_timeout_t* value, so *K_NO_WAIT* means to start the thread immediately. The corresponding parameter to *K_THREAD_DEFINE* is a duration in integral milliseconds, so the equivalent argument is 0.

User Mode Constraints This section only applies if CONFIG_USERSPACE is enabled, and a user thread tries to create a new thread. The *k_thread_create()* API is still used, but there are additional constraints which must be met or the calling thread will be terminated:

- The calling thread must have permissions granted on both the child thread and stack parameters; both are tracked by the kernel as kernel objects.
- The child thread and stack objects must be in an uninitialized state, i.e. it is not currently running and the stack memory is unused.

- The stack size parameter passed in must be equal to or less than the bounds of the stack object when it was declared.
- The *K_USER* option must be used, as user threads can only create other user threads.
- The *K_ESSENTIAL* option must not be used, user threads may not be considered essential threads.
- The priority of the child thread must be a valid priority value, and equal to or lower than the parent thread.

Dropping Permissions If CONFIG_USERSPACE is enabled, a thread running in supervisor mode may perform a one-way transition to user mode using the *k_thread_user_mode_enter()* API. This is a one-way operation which will reset and zero the thread's stack memory. The thread will be marked as non-essential.

Terminating a Thread A thread terminates itself by returning from its entry point function.

The following code illustrates the ways a thread can terminate.

```
void my_entry_point(int unused1, int unused2, int unused3)
{
    while (1) {
        ...
        if (<some condition>) {
            return; /* thread terminates from mid-entry point function */
        }
        ...
    }

    /* thread terminates at end of entry point function */
}
```

If CONFIG_USERSPACE is enabled, aborting a thread will additionally mark the thread and stack objects as uninitialized so that they may be re-used.

Runtime Statistics Thread runtime statistics can be gathered and retrieved if CONFIG_THREAD_RUNTIME_STATS is enabled, for example, total number of execution cycles of a thread.

By default, the runtime statistics are gathered using the default kernel timer. For some architectures, SoCs or boards, there are timers with higher resolution available via timing functions. Using of these timers can be enabled via CONFIG_THREAD_RUNTIME_STATS_USE_TIMING_FUNCTIONS.

Here is an example:

```
k_thread_runtime_stats_t rt_stats_thread;

k_thread_runtime_stats_get(k_current_get(), &rt_stats_thread);

printk("Cycles: %llu\n", rt_stats_thread.execution_cycles);
```

Suggested Uses Use threads to handle processing that cannot be handled in an ISR.

Use separate threads to handle logically distinct processing operations that can execute in parallel.

Configuration Options Related configuration options:

- CONFIG_MAIN_THREAD_PRIORITY
- CONFIG_MAIN_STACK_SIZE
- CONFIG_IDLE_STACK_SIZE
- CONFIG_THREAD_CUSTOM_DATA
- CONFIG_NUM_COOP_PRIORITIES
- CONFIG_NUM_PREEMPT_PRIORITIES
- CONFIG_TIMESLICING
- CONFIG_TIMESLICE_SIZE
- CONFIG_TIMESLICE_PRIORITY
- CONFIG_USERSPACE

API Reference

group thread_apis

Defines

K_ESSENTIAL

system thread that must not abort

K_FP_REGS

FPU registers are managed by context switch.

This option indicates that the thread uses the CPU's floating point registers. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. No effect if CONFIG_FPU_SHARING is not enabled.

K_USER

user mode thread

This thread has dropped from supervisor mode to user mode and consequently has additional restrictions

K_INHERIT_PERMS

Inherit Permissions.

Indicates that the thread being created should inherit all kernel object permissions from the thread that created it. No effect if CONFIG_USERSPACE is not enabled.

K_CALLBACK_STATE

Callback item state.

This is a single bit of state reserved for "callback manager" utilities (p4wq initially) who need to track operations invoked from within a user-provided callback they have been invoked. Effectively it serves as a tiny bit of zero-overhead TLS data.

k_thread_access_grant(thread, ...)

Grant a thread access to a set of kernel objects.

This is a convenience function. For the provided thread, grant access to the remaining arguments, which must be pointers to kernel objects.

The thread object must be initialized (i.e. running). The objects don't need to be. Note that NULL shouldn't be passed as an argument.

Parameters

- thread – Thread to grant access to objects
- ... – list of kernel object pointers

K_THREAD_DEFINE(name, stack_size, entry, p1, p2, p3, prio, options, delay)

Statically define and initialize a thread.

The thread may be scheduled for immediate execution or a delayed start.

Thread options are architecture-specific, and can include K_ESSENTIAL, K_FP_REGS, and K_SSE_REGS. Multiple options may be specified by separating them using "|" (the logical OR operator).

The ID of the thread can be accessed using:

```
extern const k_tid_t <name>;
```

Parameters

- name – Name of the thread.
- stack_size – Stack size in bytes.
- entry – Thread entry function.
- p1 – 1st entry point parameter.
- p2 – 2nd entry point parameter.
- p3 – 3rd entry point parameter.
- prio – Thread priority.
- options – Thread options.
- delay – Scheduling delay (in milliseconds), zero for no delay.

Typedefs

typedef void (*k_thread_user_cb_t)(const struct *k_thread* *thread, void *user_data)

Functions

void k_thread_foreach(*k_thread_user_cb_t* user_cb, void *user_data)

Iterate over all the threads in the system.

This routine iterates over all the threads in the system and calls the user_cb function for each thread.

Note: CONFIG_THREAD_MONITOR must be set for this function to be effective.

Note: This API uses *k_spin_lock* to protect the _kernel.threads list which means creation of new threads and terminations of existing threads are blocked until this API returns.

Parameters

- `user_cb` – Pointer to the user callback function.
- `user_data` – Pointer to user data.

Returns N/A

void `k_thread_foreach_unlocked`(*k_thread_user_cb_t* user_cb, void *user_data)

Iterate over all the threads in the system without locking.

This routine works exactly the same like *k_thread_foreach* but unlocks interrupts when user_cb is executed.

Note: `CONFIG_THREAD_MONITOR` must be set for this function to be effective.

Note: This API uses *k_spin_lock* only when accessing the _kernel.threads queue elements. It unlocks it during user callback function processing. If a new task is created when this `foreach` function is in progress, the added new task would not be included in the enumeration. If a task is aborted during this enumeration, there would be a race here and there is a possibility that this aborted task would be included in the enumeration.

Note: If the task is aborted and the memory occupied by its *k_thread* structure is reused when this `k_thread_foreach_unlocked` is in progress it might even lead to the system behave unstable. This function may never return, as it would follow some `next` task pointers treating given pointer as a pointer to the *k_thread* structure while it is something different right now. Do not reuse the memory that was occupied by *k_thread* structure of aborted task if it was aborted after this function was called in any context.

Parameters

- `user_cb` – Pointer to the user callback function.
- `user_data` – Pointer to user data.

k_tid_t `k_thread_create`(struct *k_thread* *new_thread, k_thread_stack_t *stack, size_t stack_size, k_thread_entry_t entry, void *p1, void *p2, void *p3, int prio, uint32_t options, *k_timeout_t* delay)

Create a thread.

This routine initializes a thread, then schedules it for execution.

The new thread may be scheduled for immediate execution or a delayed start. If the newly spawned thread does not have a delayed start the kernel scheduler may preempt the current thread to allow the new thread to execute.

Thread options are architecture-specific, and can include K_ESSENTIAL, K_FP_REGS, and K_SSE_REGS. Multiple options may be specified by separating them using "|" (the logical OR operator).

Stack objects passed to this function must be originally defined with either of these macros in order to be portable:

- *K_THREAD_STACK_DEFINE()* - For stacks that may support either user or supervisor threads.
- K_KERNEL_STACK_DEFINE() - For stacks that may support supervisor threads only. These stacks use less memory if CONFIG_USERSPACE is enabled.

The stack_size parameter has constraints. It must either be:

- The original size value passed to *K_THREAD_STACK_DEFINE()* or K_KERNEL_STACK_DEFINE()
- The return value of *K_THREAD_STACK_SIZEOF(stack)* if the stack was defined with *K_THREAD_STACK_DEFINE()*
- The return value of *K_KERNEL_STACK_SIZEOF(stack)* if the stack was defined with K_KERNEL_STACK_DEFINE().

Using other values, or sizeof(stack) may produce undefined behavior.

Parameters

- `new_thread` – Pointer to uninitialized struct *k_thread*
- `stack` – Pointer to the stack space.
- `stack_size` – Stack size in bytes.
- `entry` – Thread entry function.
- `p1` – 1st entry point parameter.
- `p2` – 2nd entry point parameter.
- `p3` – 3rd entry point parameter.
- `prio` – Thread priority.
- `options` – Thread options.
- `delay` – Scheduling delay, or K_NO_WAIT (for no delay).

Returns ID of new thread.

FUNC_NORETURN void `k_thread_user_mode_enter`(k_thread_entry_t entry, void *p1, void *p2, void *p3)

Drop a thread's privileges permanently to user mode.

This allows a supervisor thread to be re-used as a user thread. This function does not return, but control will transfer to the provided entry point as if this was a new user thread.

The implementation ensures that the stack buffer contents are erased. Any thread-local storage will be reverted to a pristine state.

Memory domain membership, resource pool assignment, kernel object permissions, priority, and thread options are preserved.

A common use of this function is to re-use the main thread as a user thread once all supervisor mode-only tasks have been completed.

Parameters

- `entry` – Function to start executing from
- `p1` – 1st entry point parameter
- `p2` – 2nd entry point parameter
- `p3` – 3rd entry point parameter

static inline void `k_thread_heap_assign`(struct *k_thread* *thread, struct *k_heap* *heap)

Assign a resource memory pool to a thread.

By default, threads have no resource pool assigned unless their parent thread has a resource pool, in which case it is inherited. Multiple threads may be assigned to the same memory pool.

Changing a thread's resource pool will not migrate allocations from the previous pool.

Parameters

- `thread` – Target thread to assign a memory pool for resource requests.
- `heap` – Heap object to use for resources, or NULL if the thread should no longer have a memory pool.

void `k_thread_system_pool_assign`(struct *k_thread* *thread)

Assign the system heap as a thread's resource pool.

Similar to z_thread_heap_assign(), but the thread will use the kernel heap to draw memory.

Use with caution, as a malicious thread could perform DoS attacks on the kernel heap.

Parameters

- `thread` – Target thread to assign the system heap for resource requests

int `k_thread_join`(struct *k_thread* *thread, *k_timeout_t* timeout)

Sleep until a thread exits.

The caller will be put to sleep until the target thread exits, either due to being aborted, self-exiting, or taking a fatal error. This API returns immediately if the thread isn't running.

This API may only be called from ISRs with a K_NO_WAIT timeout, where it can be useful as a predicate to detect when a thread has aborted.

Parameters

- `thread` – Thread to wait to exit
- `timeout` – upper bound time to wait for the thread to exit.

Return values

- `0` – success, target thread has exited or wasn't running
- `-EBUSY` – returned without waiting
- `-EAGAIN` – waiting period timed out
- `-EDEADLK` – target thread is joining on the caller, or target thread is the caller

int32_t `k_sleep`(*k_timeout_t* timeout)

Put the current thread to sleep.

This routine puts the current thread to sleep for *duration*, specified as a *k_timeout_t* object.

Note: if *timeout* is set to K_FOREVER then the thread is suspended.

Parameters

- `timeout` – Desired duration of sleep.

Returns Zero if the requested time has elapsed or the number of milliseconds left to sleep, if thread was woken up by *k_wakeup* call.

static inline int32_t k_msleep(int32_t ms)

Put the current thread to sleep.

This routine puts the current thread to sleep for *duration* milliseconds.

Parameters

- ms – Number of milliseconds to sleep.

Returns Zero if the requested time has elapsed or the number of milliseconds left to sleep, if thread was woken up by *k_wakeup* call.

int32_t k_usleep(int32_t us)

Put the current thread to sleep with microsecond resolution.

This function is unlikely to work as expected without kernel tuning. In particular, because the lower bound on the duration of a sleep is the duration of a tick, CONFIG_SYS_CLOCK_TICKS_PER_SEC must be adjusted to achieve the resolution desired. The implications of doing this must be understood before attempting to use *k_usleep()*. Use with caution.

Parameters

- us – Number of microseconds to sleep.

Returns Zero if the requested time has elapsed or the number of microseconds left to sleep, if thread was woken up by *k_wakeup* call.

void k_busy_wait(uint32_t usec_to_wait)

Cause the current thread to busy wait.

This routine causes the current thread to execute a "do nothing" loop for *usec_to_wait* microseconds.

Note: The clock used for the microsecond-resolution delay here may be skewed relative to the clock used for system timeouts like *k_sleep()*. For example k_busy_wait(1000) may take slightly more or less time than k_sleep(K_MSEC(1)), with the offset dependent on clock tolerances.

Returns N/A

void k_yield(void)

Yield the current thread.

This routine causes the current thread to yield execution to another thread of the same or higher priority. If there are no other ready threads of the same or higher priority, the routine returns immediately.

Returns N/A

void k_wakeup(k_tid_t thread)

Wake up a sleeping thread.

This routine prematurely wakes up *thread* from sleeping.

If *thread* is not currently sleeping, the routine has no effect.

Parameters

- thread – ID of thread to wake.

Returns N/A

__attribute_const__ static inline k_tid_t `k_current_get`(void)

Get thread ID of the current thread.

Returns ID of current thread.

void `k_thread_abort`(k_tid_t thread)

Abort a thread.

This routine permanently stops execution of *thread*. The thread is taken off all kernel queues it is part of (i.e. the ready queue, the timeout queue, or a kernel object wait queue). However, any kernel resources the thread might currently own (such as mutexes or memory blocks) are not released. It is the responsibility of the caller of this routine to ensure all necessary cleanup is performed.

After *k_thread_abort()* returns, the thread is guaranteed not to be running or to become runnable anywhere on the system. Normally this is done via blocking the caller (in the same manner as *k_thread_join()*), but in interrupt context on SMP systems the implementation is required to spin for threads that are running on other CPUs. Note that as specified, this means that on SMP platforms it is possible for application code to create a deadlock condition by simultaneously aborting a cycle of threads using at least one termination from interrupt context. Zephyr cannot detect all such conditions.

Parameters

- `thread` – ID of thread to abort.

Returns N/A

void `k_thread_start`(k_tid_t thread)

Start an inactive thread.

If a thread was created with K_FOREVER in the delay parameter, it will not be added to the scheduling queue until this function is called on it.

Parameters

- `thread` – thread to start

k_ticks_t `k_thread_timeout_expires_ticks`(const struct *k_thread* *t)

Get time when a thread wakes up, in system ticks.

This routine computes the system uptime when a waiting thread next executes, in units of system ticks. If the thread is not waiting, it returns current system time.

k_ticks_t `k_thread_timeout_remaining_ticks`(const struct *k_thread* *t)

Get time remaining before a thread wakes up, in system ticks.

This routine computes the time remaining before a waiting thread next executes, in units of system ticks. If the thread is not waiting, it returns zero.

int `k_thread_priority_get`(k_tid_t thread)

Get a thread's priority.

This routine gets the priority of *thread*.

Parameters

- `thread` – ID of thread whose priority is needed.

Returns Priority of *thread*.

void `k_thread_priority_set`(k_tid_t thread, int prio)

Set a thread's priority.

This routine immediately changes the priority of *thread*.

Rescheduling can occur immediately depending on the priority *thread* is set to:

- If its priority is raised above the priority of the caller of this function, and the caller is preemptible, *thread* will be scheduled in.
- If the caller operates on itself, it lowers its priority below that of other threads in the system, and the caller is preemptible, the thread of highest priority will be scheduled in.

Priority can be assigned in the range of -CONFIG_NUM_COOP_PRIORITIES to CONFIG_NUM_PREEMPT_PRIORITIES-1, where -CONFIG_NUM_COOP_PRIORITIES is the highest priority.

Warning: Changing the priority of a thread currently involved in mutex priority inheritance may result in undefined behavior.

Parameters

- `thread` – ID of thread whose priority is to be set.
- `prio` – New priority.

Returns N/A

void `k_thread_deadline_set`(k_tid_t thread, int deadline)

Set deadline expiration time for scheduler.

This sets the "deadline" expiration as a time delta from the current time, in the same units used by *k_cycle_get_32()*. The scheduler (when deadline scheduling is enabled) will choose the next expiring thread when selecting between threads at the same static priority. Threads at different priorities will be scheduled according to their static priority.

Note: Deadlines are stored internally using 32 bit unsigned integers. The number of cycles between the "first" deadline in the scheduler queue and the "last" deadline must be less than 2^31 (i.e a signed non-negative quantity). Failure to adhere to this rule may result in scheduled threads running in an incorrect deadline order.

Note: Despite the API naming, the scheduler makes no guarantees the the thread WILL be scheduled within that deadline, nor does it take extra metadata (like e.g. the "runtime" and "period" parameters in Linux sched_setattr()) that allows the kernel to validate the scheduling for achievability. Such features could be implemented above this call, which is simply input to the priority selection logic.

Note: You should enable `CONFIG_SCHED_DEADLINE` in your project configuration.

Parameters

- `thread` – A thread on which to set the deadline
- `deadline` – A time delta, in cycle units

int `k_thread_cpu_mask_clear`(k_tid_t thread)

Sets all CPU enable masks to zero.

After this returns, the thread will no longer be schedulable on any CPUs. The thread must not be currently runnable.

Note: You should enable `CONFIG_SCHED_DEADLINE` in your project configuration.

Parameters

- `thread` – Thread to operate upon

Returns Zero on success, otherwise error code

int `k_thread_cpu_mask_enable_all`(k_tid_t thread)

Sets all CPU enable masks to one.

After this returns, the thread will be schedulable on any CPU. The thread must not be currently runnable.

Note: You should enable `CONFIG_SCHED_DEADLINE` in your project configuration.

Parameters

- `thread` – Thread to operate upon

Returns Zero on success, otherwise error code

int `k_thread_cpu_mask_enable`(k_tid_t thread, int cpu)

Enable thread to run on specified CPU.

The thread must not be currently runnable.

Note: You should enable `CONFIG_SCHED_DEADLINE` in your project configuration.

Parameters

- `thread` – Thread to operate upon
- `cpu` – CPU index

Returns Zero on success, otherwise error code

int `k_thread_cpu_mask_disable`(k_tid_t thread, int cpu)

Prevent thread to run on specified CPU.

The thread must not be currently runnable.

Note: You should enable `CONFIG_SCHED_DEADLINE` in your project configuration.

Parameters

- `thread` – Thread to operate upon
- `cpu` – CPU index

Returns Zero on success, otherwise error code

void `k_thread_suspend`(k_tid_t thread)

Suspend a thread.

This routine prevents the kernel scheduler from making *thread* the current thread. All other internal operations on *thread* are still performed; for example, kernel objects it is waiting on are still handed to it. Note that any existing timeouts (e.g. *k_sleep()*, or a timeout argument to *k_sem_take()* et. al.) will be canceled. On resume, the thread will begin running immediately and return from the blocked call.

If *thread* is already suspended, the routine has no effect.

Parameters

- `thread` – ID of thread to suspend.

Returns N/A

void `k_thread_resume`(k_tid_t thread)

Resume a suspended thread.

This routine allows the kernel scheduler to make *thread* the current thread, when it is next eligible for that role.

If *thread* is not currently suspended, the routine has no effect.

Parameters

- `thread` – ID of thread to resume.

Returns N/A

void `k_sched_time_slice_set`(int32_t slice, int prio)

Set time-slicing period and scope.

This routine specifies how the scheduler will perform time slicing of preemptible threads.

To enable time slicing, *slice* must be non-zero. The scheduler ensures that no thread runs for more than the specified time limit before other threads of that priority are given a chance to execute. Any thread whose priority is higher than *prio* is exempted, and may execute as long as desired without being preempted due to time slicing.

Time slicing only limits the maximum amount of time a thread may continuously execute. Once the scheduler selects a thread for execution, there is no minimum guaranteed time the thread will execute before threads of greater or equal priority are scheduled.

When the current thread is the only one of that priority eligible for execution, this routine has no effect; the thread is immediately rescheduled after the slice period expires.

To disable timeslicing, set both *slice* and *prio* to zero.

Parameters

- `slice` – Maximum time slice length (in milliseconds).
- `prio` – Highest thread priority level eligible for time slicing.

Returns N/A

void `k_sched_lock`(void)

Lock the scheduler.

This routine prevents the current thread from being preempted by another thread by instructing the scheduler to treat it as a cooperative thread. If the thread subsequently performs an operation that makes it unready, it will be context switched out in the normal manner. When the thread again becomes the current thread, its non-preemptible status is maintained.

This routine can be called recursively.

Note: *k_sched_lock()* and *k_sched_unlock()* should normally be used when the operation being performed can be safely interrupted by ISRs. However, if the amount of processing involved is very small, better performance may be obtained by using *irq_lock()* and *irq_unlock()*.

Returns N/A

void k_sched_unlock(void)

Unlock the scheduler.

This routine reverses the effect of a previous call to *k_sched_lock()*. A thread must call the routine once for each time it called *k_sched_lock()* before the thread becomes preemptible.

Returns N/A

void k_thread_custom_data_set(void *value)

Set current thread's custom data.

This routine sets the custom data for the current thread to @ value.

Custom data is not used by the kernel itself, and is freely available for a thread to use as it sees fit. It can be used as a framework upon which to build thread-local storage.

Parameters

- value – New custom data value.

Returns N/A

void *k_thread_custom_data_get(void)

Get current thread's custom data.

This routine returns the custom data for the current thread.

Returns Current custom data value.

int k_thread_name_set(k_tid_t thread, const char *str)

Set current thread name.

Set the name of the thread to be used when CONFIG_THREAD_MONITOR is enabled for tracing and debugging.

Parameters

- thread – Thread to set name, or NULL to set the current thread
- str – Name string

Return values

- 0 – on success
- -EFAULT – Memory access error with supplied string
- -ENOSYS – Thread name configuration option not enabled
- -EINVAL – Thread name too long

const char *k_thread_name_get(k_tid_t thread)

Get thread name.

Get the name of a thread

Parameters

- thread – Thread ID

Return values Thread – name, or NULL if configuration not enabled

int k_thread_name_copy(k_tid_t thread, char *buf, size_t size)

Copy the thread name into a supplied buffer.

Parameters

- thread – Thread to obtain name information
- buf – Destination buffer
- size – Destination buffer size

Return values

- `-ENOSPC` – Destination buffer too small
- `-EFAULT` – Memory access error
- `-ENOSYS` – Thread name feature not enabled
- `0` – Success

const char *`k_thread_state_str`(k_tid_t thread_id)

Get thread state string.

Get the human friendly thread state string

Parameters

- `thread_id` – Thread ID

Return values `Thread` – state string, empty if no state flag is set

struct `k_thread`

#include <thread.h> Thread Structure

Public Members

struct _callee_saved `callee_saved`

defined by the architecture, but all archs need these

void *`init_data`

static thread init data

_wait_q_t `join_queue`

threads waiting in *k_thread_join()*

struct __thread_entry `entry`

thread entry and parameters description

struct *k_thread* *`next_thread`

next item in list of all threads

void *`custom_data`

crude thread-local storage

struct _thread_stack_info `stack_info`

Stack Info

struct _mem_domain_info `mem_domain_info`

memory domain info of the thread

k_thread_stack_t *`stack_obj`

Base address of thread stack

void *`syscall_frame`

current syscall frame pointer

int `swap_retval`

z_swap() return value

void *`switch_handle`

Context handle returned via *arch_switch()*

struct *k_heap* *`resource_pool`

resource pool

struct _thread_arch `arch`

arch-specifics: must always be at the end

group `thread_stack_api`

Thread Stack APIs.

Defines

`K_KERNEL_STACK_ARRAY_EXTERN`(sym, nmemb, size)

Obtain an extern reference to a stack array.

This macro properly brings the symbol of a stack array declared elsewhere into scope.

Parameters

- `sym` – Thread stack symbol name
- `nmemb` – Number of stacks to declare
- `size` – Size of the stack memory region

`K_KERNEL_PINNED_STACK_ARRAY_EXTERN`(sym, nmemb, size)

Obtain an extern reference to a pinned stack array.

This macro properly brings the symbol of a pinned stack array declared elsewhere into scope.

Parameters

- `sym` – Thread stack symbol name
- `nmemb` – Number of stacks to declare
- `size` – Size of the stack memory region

`K_KERNEL_STACK_DEFINE`(sym, size)

Define a toplevel kernel stack memory region.

This declares a region of memory for use as a thread stack, for threads that exclusively run in supervisor mode. This is also suitable for declaring special stacks for interrupt or exception handling.

Stacks declared with this macro may not host user mode threads.

It is legal to precede this definition with the 'static' keyword.

It is NOT legal to take the sizeof(sym) and pass that to the stackSize parameter of *k_thread_create()*, it may not be the same as the 'size' parameter. Use *K_KERNEL_STACK_SIZEOF()* instead.

The total amount of memory allocated may be increased to accommodate fixed-size stack overflow guards.

Parameters

- `sym` – Thread stack symbol name
- `size` – Size of the stack memory region

`K_KERNEL_PINNED_STACK_DEFINE(sym, size)`

Define a toplevel kernel stack memory region in pinned section.

See *K_KERNEL_STACK_DEFINE()* for more information and constraints.

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_KERNEL_STACK_DEFINE()*.

Parameters

- `sym` – Thread stack symbol name
- `size` – Size of the stack memory region

`K_KERNEL_STACK_ARRAY_DEFINE(sym, nmemb, size)`

Define a toplevel array of kernel stack memory regions.

Stacks declared with this macro may not host user mode threads.

Parameters

- `sym` – Kernel stack array symbol name
- `nmemb` – Number of stacks to declare
- `size` – Size of the stack memory region

`K_KERNEL_PINNED_STACK_ARRAY_DEFINE(sym, nmemb, size)`

Define a toplevel array of kernel stack memory regions in pinned section.

See *K_KERNEL_STACK_ARRAY_DEFINE()* for more information and constraints.

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_KERNEL_STACK_ARRAY_DEFINE()*.

Parameters

- `sym` – Kernel stack array symbol name
- `nmemb` – Number of stacks to declare
- `size` – Size of the stack memory region

`K_KERNEL_STACK_MEMBER(sym, size)`

Declare an embedded stack memory region.

Used for kernel stacks embedded within other data structures.

Stacks declared with this macro may not host user mode threads.

Parameters

- `sym` – Thread stack symbol name
- `size` – Size of the stack memory region

`K_KERNEL_STACK_SIZEOF(sym)`

`K_THREAD_STACK_SIZEOF(sym)`

Return the size in bytes of a stack memory region.

Convenience macro for passing the desired stack size to *k_thread_create()* since the underlying implementation may actually create something larger (for instance a guard area).

The value returned here is not guaranteed to match the 'size' parameter passed to K_THREAD_STACK_DEFINE and may be larger, but is always safe to pass to *k_thread_create()* for the associated stack object.

Parameters

- `sym` – Stack memory symbol

Returns Size of the stack buffer

`K_THREAD_STACK_DEFINE(sym, size)`

Declare a toplevel thread stack memory region.

This declares a region of memory suitable for use as a thread's stack.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

The declared symbol will always be a k_thread_stack_t which can be passed to *k_thread_create()*, but should otherwise not be manipulated. If the buffer inside needs to be examined, examine thread->stack_info for the associated thread object to obtain the boundaries.

It is legal to precede this definition with the 'static' keyword.

It is NOT legal to take the sizeof(sym) and pass that to the stackSize parameter of *k_thread_create()*, it may not be the same as the 'size' parameter. Use *K_THREAD_STACK_SIZEOF()* instead.

Some arches may round the size of the usable stack region up to satisfy alignment constraints. *K_THREAD_STACK_SIZEOF()* will return the aligned size.

Parameters

- `sym` – Thread stack symbol name
- `size` – Size of the stack memory region

`K_THREAD_PINNED_STACK_DEFINE(sym, size)`

Define a toplevel thread stack memory region in pinned section.

This declares a region of memory suitable for use as a thread's stack.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

The declared symbol will always be a k_thread_stack_t which can be passed to *k_thread_create()*, but should otherwise not be manipulated. If the buffer inside needs to be examined, examine thread->stack_info for the associated thread object to obtain the boundaries.

It is legal to precede this definition with the 'static' keyword.

It is NOT legal to take the sizeof(sym) and pass that to the stackSize parameter of *k_thread_create()*, it may not be the same as the 'size' parameter. Use *K_THREAD_STACK_SIZEOF()* instead.

Some arches may round the size of the usable stack region up to satisfy alignment constraints. *K_THREAD_STACK_SIZEOF()* will return the aligned size.

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_THREAD_STACK_DEFINE()*.

Parameters

- `sym` – Thread stack symbol name
- `size` – Size of the stack memory region

K_THREAD_STACK_LEN(size)

Calculate size of stacks to be allocated in a stack array.

This macro calculates the size to be allocated for the stacks inside a stack array. It accepts the indicated "size" as a parameter and if required, pads some extra bytes (e.g. for MPU scenarios). Refer K_THREAD_STACK_ARRAY_DEFINE definition to see how this is used. The returned size ensures each array member will be aligned to the required stack base alignment.

Parameters

- size – Size of the stack memory region

Returns Appropriate size for an array member

K_THREAD_STACK_ARRAY_DEFINE(sym, nmemb, size)

Declare a toplevel array of thread stack memory regions.

Create an array of equally sized stacks. See K_THREAD_STACK_DEFINE definition for additional details and constraints.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

Parameters

- sym – Thread stack symbol name
- nmemb – Number of stacks to declare
- size – Size of the stack memory region

K_THREAD_PINNED_STACK_ARRAY_DEFINE(sym, nmemb, size)

Declare a toplevel array of thread stack memory regions in pinned section.

Create an array of equally sized stacks. See K_THREAD_STACK_DEFINE definition for additional details and constraints.

This is the generic, historical definition. Align to Z_THREAD_STACK_OBJ_ALIGN and put in 'noinit' section so that it isn't zeroed at boot

This puts the stack into the pinned noinit linker section if CONFIG_LINKER_USE_PINNED_SECTION is enabled, or else it would put the stack into the same section as *K_THREAD_STACK_DEFINE()*.

Parameters

- sym – Thread stack symbol name
- nmemb – Number of stacks to declare
- size – Size of the stack memory region

K_THREAD_STACK_MEMBER(sym, size)

Declare an embedded stack memory region.

Used for stacks embedded within other data structures. Use is highly discouraged but in some cases necessary. For memory protection scenarios, it is very important that any RAM preceding this member not be writable by threads else a stack overflow will lead to silent corruption. In other words, the containing data structure should live in RAM owned by the kernel.

A user thread can only be started with a stack defined in this way if the thread starting it is in supervisor mode.

This is now deprecated, as stacks defined in this way are not usable from user mode. Use K_KERNEL_STACK_MEMBER.

Parameters

- sym – Thread stack symbol name

- `size` – Size of the stack memory region

Scheduling

The kernel's priority-based scheduler allows an application's threads to share the CPU.

Concepts The scheduler determines which thread is allowed to execute at any point in time; this thread is known as the **current thread**.

There are various points in time when the scheduler is given an opportunity to change the identity of the current thread. These points are called **reschedule points**. Some potential reschedule points are:

- transition of a thread from running state to a suspended or waiting state, for example by *k_sem_take()* or *k_sleep()*.
- transition of a thread to the *ready state*, for example by *k_sem_give()* or *k_thread_start()*
- return to thread context after processing an interrupt
- when a running thread invokes *k_yield()*

A thread **sleeps** when it voluntarily initiates an operation that transitions itself to a suspended or waiting state.

Whenever the scheduler changes the identity of the current thread, or when execution of the current thread is replaced by an ISR, the kernel first saves the current thread's CPU register values. These register values get restored when the thread later resumes execution.

Scheduling Algorithm The kernel's scheduler selects the highest priority ready thread to be the current thread. When multiple ready threads of the same priority exist, the scheduler chooses the one that has been waiting longest.

A thread's relative priority is primarily determined by its static priority. However, when both earliest-deadline-first scheduling is enabled (`CONFIG_SCHED_DEADLINE`) and a choice of threads have equal static priority, then the thread with the earlier deadline is considered to have the higher priority. Thus, when earliest-deadline-first scheduling is enabled, two threads are only considered to have the same priority when both their static priorities and deadlines are equal. The routine *k_thread_deadline_set()* is used to set a thread's deadline.

Note: Execution of ISRs takes precedence over thread execution, so the execution of the current thread may be replaced by an ISR at any time unless interrupts have been masked. This applies to both cooperative threads and preemptive threads.

The kernel can be built with one of several choices for the ready queue implementation, offering different choices between code size, constant factor runtime overhead and performance scaling when many threads are added.

- Simple linked-list ready queue (`CONFIG_SCHED_DUMB`)

  The scheduler ready queue will be implemented as a simple unordered list, with very fast constant time performance for single threads and very low code size. This implementation should be selected on systems with constrained code size that will never see more than a small number (3, maybe) of runnable threads in the queue at any given time. On most platforms (that are not otherwise using the red/black tree) this results in a savings of ~2k of code size.

- Red/black tree ready queue (`CONFIG_SCHED_SCALABLE`)

  The scheduler ready queue will be implemented as a red/black tree. This has rather slower constant-time insertion and removal overhead, and on most platforms (that are not otherwise using the red/black tree somewhere) requires an extra ~2kb of code. The resulting behavior will scale cleanly and quickly into the many thousands of threads.

Use this for applications needing many concurrent runnable threads (> 20 or so). Most applications won't need this ready queue implementation.

- Traditional multi-queue ready queue (`CONFIG_SCHED_MULTIQ`)

  When selected, the scheduler ready queue will be implemented as the classic/textbook array of lists, one per priority (max 32 priorities).

  This corresponds to the scheduler algorithm used in Zephyr versions prior to 1.12.

  It incurs only a tiny code size overhead vs. the "dumb" scheduler and runs in O(1) time in almost all circumstances with very low constant factor. But it requires a fairly large RAM budget to store those list heads, and the limited features make it incompatible with features like deadline scheduling that need to sort threads more finely, and SMP affinity which need to traverse the list of threads.

  Typical applications with small numbers of runnable threads probably want the DUMB scheduler.

The wait_q abstraction used in IPC primitives to pend threads for later wakeup shares the same backend data structure choices as the scheduler, and can use the same options.

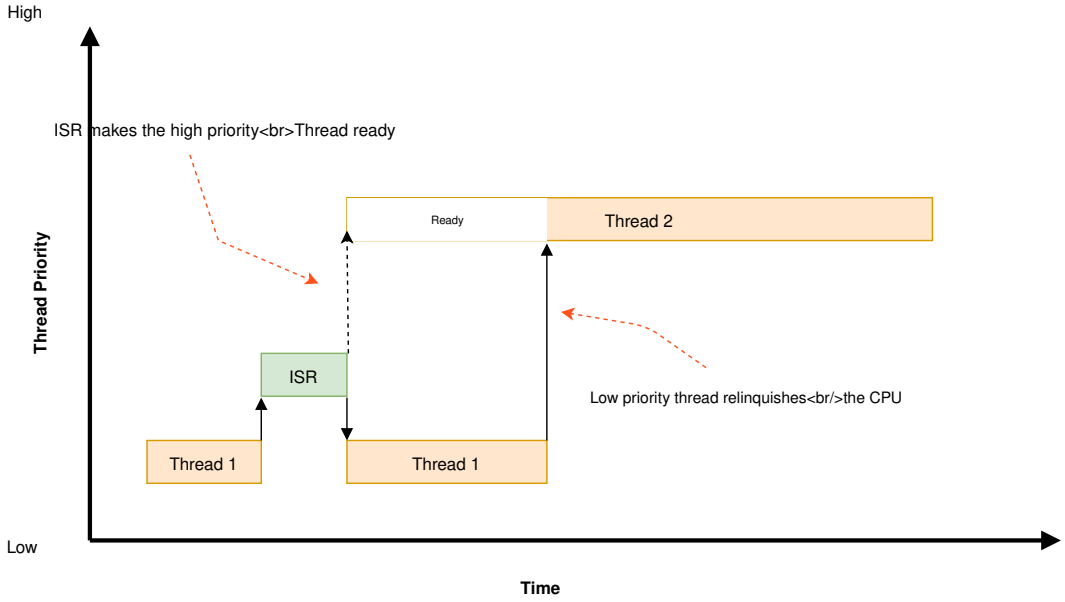
- Scalable wait_q implementation (`CONFIG_WAITQ_SCALABLE`)

  When selected, the wait_q will be implemented with a balanced tree. Choose this if you expect to have many threads waiting on individual primitives. There is a ~2kb code size increase over `CONFIG_WAITQ_DUMB` (which may be shared with `CONFIG_SCHED_SCALABLE`) if the red/black tree is not used elsewhere in the application, and pend/unpend operations on "small" queues will be somewhat slower (though this is not generally a performance path).

- Simple linked-list wait_q (`CONFIG_WAITQ_DUMB`)

  When selected, the wait_q will be implemented with a doubly-linked list. Choose this if you expect to have only a few threads blocked on any single IPC primitive.

Cooperative Time Slicing Once a cooperative thread becomes the current thread, it remains the current thread until it performs an action that makes it unready. Consequently, if a cooperative thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of higher priority and equal priority.



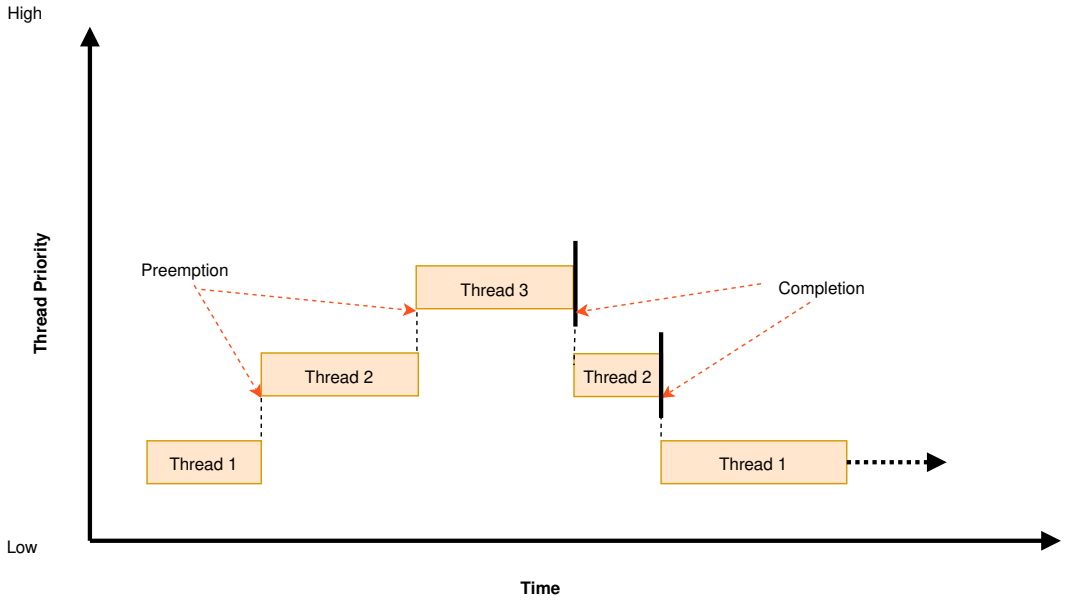
To overcome such problems, a cooperative thread can voluntarily relinquish the CPU from time to time to permit other threads to execute. A thread can relinquish the CPU in two ways:

- Calling *k_yield()* puts the thread at the back of the scheduler's prioritized list of ready threads, and then invokes the scheduler. All ready threads whose priority is higher or equal to that of the yielding thread are then allowed to execute before the yielding thread is rescheduled. If no such

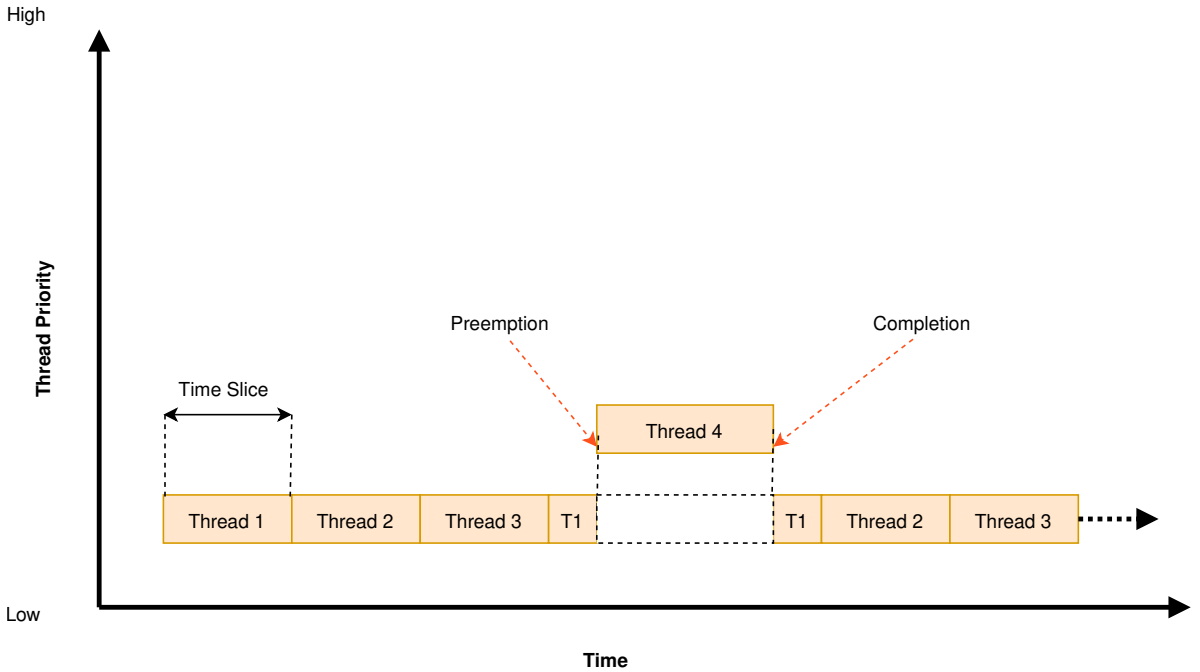
ready threads exist, the scheduler immediately reschedules the yielding thread without context switching.

- Calling *k_sleep()* makes the thread unready for a specified time period. Ready threads of *all* priorities are then allowed to execute; however, there is no guarantee that threads whose priority is lower than that of the sleeping thread will actually be scheduled before the sleeping thread becomes ready once again.

Preemptive Time Slicing Once a preemptive thread becomes the current thread, it remains the current thread until a higher priority thread becomes ready, or until the thread performs an action that makes it unready. Consequently, if a preemptive thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of equal priority.



To overcome such problems, a preemptive thread can perform cooperative time slicing (as described above), or the scheduler's time slicing capability can be used to allow other threads of the same priority to execute.



The scheduler divides time into a series of **time slices**, where slices are measured in system clock ticks. The time slice size is configurable, but this size can be changed while the application is running.

At the end of every time slice, the scheduler checks to see if the current thread is preemptible and, if so, implicitly invokes *k_yield()* on behalf of the thread. This gives other ready threads of the same priority the opportunity to execute before the current thread is scheduled again. If no threads of equal priority are ready, the current thread remains the current thread.

Threads with a priority higher than specified limit are exempt from preemptive time slicing, and are never preempted by a thread of equal priority. This allows an application to use preemptive time slicing only when dealing with lower priority threads that are less time-sensitive.

Note: The kernel's time slicing algorithm does *not* ensure that a set of equal-priority threads receive an equitable amount of CPU time, since it does not measure the amount of time a thread actually gets to execute. However, the algorithm *does* ensure that a thread never executes for longer than a single time slice without being required to yield.

Scheduler Locking A preemptible thread that does not wish to be preempted while performing a critical operation can instruct the scheduler to temporarily treat it as a cooperative thread by calling *k_sched_lock()*. This prevents other threads from interfering while the critical operation is being performed.

Once the critical operation is complete the preemptible thread must call *k_sched_unlock()* to restore its normal, preemptible status.

If a thread calls *k_sched_lock()* and subsequently performs an action that makes it unready, the scheduler will switch the locking thread out and allow other threads to execute. When the locking thread again becomes the current thread, its non-preemptible status is maintained.

Note: Locking out the scheduler is a more efficient way for a preemptible thread to prevent preemption than changing its priority level to a negative value.

Thread Sleeping A thread can call *k_sleep()* to delay its processing for a specified time period. During the time the thread is sleeping the CPU is relinquished to allow other ready threads to execute. Once the specified delay has elapsed the thread becomes ready and is eligible to be scheduled once again.

A sleeping thread can be woken up prematurely by another thread using *k_wakeup()*. This technique can sometimes be used to permit the secondary thread to signal the sleeping thread that something has occurred *without* requiring the threads to define a kernel synchronization object, such as a semaphore. Waking up a thread that is not sleeping is allowed, but has no effect.

Busy Waiting A thread can call *k_busy_wait()* to perform a `busy wait` that delays its processing for a specified time period *without* relinquishing the CPU to another ready thread.

A busy wait is typically used instead of thread sleeping when the required delay is too short to warrant having the scheduler context switch from the current thread to another thread and then back again.

Suggested Uses Use cooperative threads for device drivers and other performance-critical work.

Use cooperative threads to implement mutually exclusion without the need for a kernel object, such as a mutex.

Use preemptive threads to give priority to time-sensitive processing over less time-sensitive processing.

System Threads

A *system thread* is a thread that the kernel spawns automatically during system initialization.

The kernel spawns the following system threads:

Main thread This thread performs kernel initialization, then calls the application's `main()` function (if one is defined).

By default, the main thread uses the highest configured preemptible thread priority (i.e. 0). If the kernel is not configured to support preemptible threads, the main thread uses the lowest configured cooperative thread priority (i.e. -1).

The main thread is an essential thread while it is performing kernel initialization or executing the application's `main()` function; this means a fatal system error is raised if the thread aborts. If `main()` is not defined, or if it executes and then does a normal return, the main thread terminates normally and no error is raised.

Idle thread This thread executes when there is no other work for the system to do. If possible, the idle thread activates the board's power management support to save power; otherwise, the idle thread simply performs a "do nothing" loop. The idle thread remains in existence as long as the system is running and never terminates.

The idle thread always uses the lowest configured thread priority. If this makes it a cooperative thread, the idle thread repeatedly yields the CPU to allow the application's other threads to run when they need to.

The idle thread is an essential thread, which means a fatal system error is raised if the thread aborts.

Additional system threads may also be spawned, depending on the kernel and board configuration options specified by the application. For example, enabling the system workqueue spawns a system thread that services the work items submitted to it. (See *Workqueue Threads*.)

Implementation

Writing a main() function

An application-supplied `main()` function begins executing once kernel initialization is complete. The kernel does not pass any arguments to the function.

The following code outlines a trivial `main()` function. The function used by a real application can be as complex as needed.

```
void main(void)
{
    /* initialize a semaphore */
    ...

    /* register an ISR that gives the semaphore */
    ...

    /* monitor the semaphore forever */
    while (1) {
        /* wait for the semaphore to be given by the ISR */
        ...
```

(continues on next page)

(continued from previous page)

```
        /* do whatever processing is now needed */
        ...
    }
}
```

Suggested Uses Use the main thread to perform thread-based processing in an application that only requires a single thread, rather than defining an additional application-specific thread.

Workqueue Threads

A *workqueue* is a kernel object that uses a dedicated thread to process work items in a first in, first out manner. Each work item is processed by calling the function specified by the work item. A workqueue is typically used by an ISR or a high-priority thread to offload non-urgent processing to a lower-priority thread so it does not impact time-sensitive processing.

Any number of workqueues can be defined (limited only by available RAM). Each workqueue is referenced by its memory address.

A workqueue has the following key properties:

- A **queue** of work items that have been added, but not yet processed.
- A **thread** that processes the work items in the queue. The priority of the thread is configurable, allowing it to be either cooperative or preemptive as required.

Regardless of workqueue thread priority the workqueue thread will yield between each submitted work item, to prevent a cooperative workqueue from starving other threads.

A workqueue must be initialized before it can be used. This sets its queue to empty and spawns the workqueue's thread. The thread runs forever, but sleeps when no work items are available.

Note: The behavior described here is changed from the Zephyr workqueue implementation used prior to release 2.6. Among the changes are:

- Precise tracking of the status of cancelled work items, so that the caller need not be concerned that an item may be processing when the cancellation returns. Checking of return values on cancellation is still required.
- Direct submission of delayable work items to the queue with `K_NO_WAIT` rather than always going through the timeout API, which could introduce delays.
- The ability to wait until a work item has completed or a queue has been drained.

- Finer control of behavior when scheduling a delayable work item, specifically allowing a previous deadline to remain unchanged when a work item is scheduled again.
- Safe handling of work item resubmission when the item is being processed on another workqueue.

Using the return values of *k_work_busy_get()* or *k_work_is_pending()*, or measurements of remaining time until delayable work is scheduled, should be avoided to prevent race conditions of the type observed with the previous implementation. See also *Workqueue Best Practices*.

Work Item Lifecycle Any number of **work items** can be defined. Each work item is referenced by its memory address.

A work item is assigned a **handler function**, which is the function executed by the workqueue's thread when the work item is processed. This function accepts a single argument, which is the address of the work item itself. The work item also maintains information about its status.

A work item must be initialized before it can be used. This records the work item's handler function and marks it as not pending.

A work item may be **queued** (*K_WORK_QUEUED*) by submitting it to a workqueue by an ISR or a thread. Submitting a work item appends the work item to the workqueue's queue. Once the workqueue's thread has processed all of the preceding work items in its queue the thread will remove the next work item from the queue and invoke the work item's handler function. Depending on the scheduling priority of the workqueue's thread, and the work required by other items in the queue, a queued work item may be processed quickly or it may remain in the queue for an extended period of time.

A delayable work item may be **scheduled** (*K_WORK_DELAYED*) to a workqueue; see *Delayable Work*.

A work item will be **running** (*K_WORK_RUNNING*) when it is running on a work queue, and may also be **canceling** (*K_WORK_CANCELING*) if it started running before a thread has requested that it be cancelled.

A work item can be in multiple states; for example it can be:

- running on a queue;
- marked canceling (because a thread used *k_work_cancel_sync()* to wait until the work item completed);
- queued to run again on the same queue;
- scheduled to be submitted to a (possibly different) queue

all simultaneously. A work item that is in any of these states is **pending** (*k_work_is_pending()*) or **busy** (*k_work_busy_get()*).

A handler function can use any kernel API available to threads. However, operations that are potentially blocking (e.g. taking a semaphore) must be used with care, since the workqueue cannot process subsequent work items in its queue until the handler function finishes executing.

The single argument that is passed to a handler function can be ignored if it is not required. If the handler function requires additional information about the work it is to perform, the work item can be embedded in a larger data structure. The handler function can then use the argument value to compute the address of the enclosing data structure with *CONTAINER_OF*, and thereby obtain access to the additional information it needs.

A work item is typically initialized once and then submitted to a specific workqueue whenever work needs to be performed. If an ISR or a thread attempts to submit a work item that is already queued the work item is not affected; the work item remains in its current place in the workqueue's queue, and the work is only performed once.

A handler function is permitted to re-submit its work item argument to the workqueue, since the work item is no longer queued at that time. This allows the handler to execute work in stages, without unduly delaying the processing of other work items in the workqueue's queue.

Important: A pending work item *must not* be altered until the item has been processed by the workqueue thread. This means a work item must not be re-initialized while it is busy. Furthermore, any additional information the work item's handler function needs to perform its work must not be altered until the handler function has finished executing.

Delayable Work An ISR or a thread may need to schedule a work item that is to be processed only after a specified period of time, rather than immediately. This can be done by **scheduling** a **delayable work item** to be submitted to a workqueue at a future time.

A delayable work item contains a standard work item but adds fields that record when and where the item should be submitted.

A delayable work item is initialized and scheduled to a workqueue in a similar manner to a standard work item, although different kernel APIs are used. When the schedule request is made the kernel initiates a timeout mechanism that is triggered after the specified delay has elapsed. Once the timeout occurs the kernel submits the work item to the specified workqueue, where it remains queued until it is processed in the standard manner.

Note that work handler used for delayable still receives a pointer to the underlying non-delayable work structure, which is not publicly accessible from *k_work_delayable*. To get access to an object that contains the delayable work object use this idiom:

```
static void work_handler(struct k_work *work)
{
        struct k_work_delayable *dwork = k_work_delayable_from_work(work);
        struct work_context *ctx = CONTAINER_OF(dwork, struct work_context,
                                                timed_work);
        ...
```

Triggered Work The *k_work_poll_submit()* interface schedules a triggered work item in response to a **poll event** (see *Polling API*), that will call a user-defined function when a monitored resource becomes available or poll signal is raised, or a timeout occurs. In contrast to *k_poll()*, the triggered work does not require a dedicated thread waiting or actively polling for a poll event.

A triggered work item is a standard work item that has the following added properties:

- A pointer to an array of poll events that will trigger work item submissions to the workqueue
- A size of the array containing poll events.

A triggered work item is initialized and submitted to a workqueue in a similar manner to a standard work item, although dedicated kernel APIs are used. When a submit request is made, the kernel begins observing kernel objects specified by the poll events. Once at least one of the observed kernel object's changes state, the work item is submitted to the specified workqueue, where it remains queued until it is processed in the standard manner.

Important: The triggered work item as well as the referenced array of poll events have to be valid and cannot be modified for a complete triggered work item lifecycle, from submission to work item execution or cancellation.

An ISR or a thread may **cancel** a triggered work item it has submitted as long as it is still waiting for a poll event. In such case, the kernel stops waiting for attached poll events and the specified work is not executed. Otherwise the cancellation cannot be performed.

System Workqueue The kernel defines a workqueue known as the *system workqueue*, which is available to any application or kernel code that requires workqueue support. The system workqueue is optional, and only exists if the application makes use of it.

Important: Additional workqueues should only be defined when it is not possible to submit new work items to the system workqueue, since each new workqueue incurs a significant cost in memory footprint. A new workqueue can be justified if it is not possible for its work items to co-exist with existing system workqueue work items without an unacceptable impact; for example, if the new work items perform blocking operations that would delay other system workqueue processing to an unacceptable degree.

How to Use Workqueues

Defining and Controlling a Workqueue A workqueue is defined using a variable of type *k_work_q*. The workqueue is initialized by defining the stack area used by its thread, initializing the *k_work_q*, either zeroing its memory or calling *k_work_queue_init()*, and then calling *k_work_queue_start()*. The stack area must be defined using *K_THREAD_STACK_DEFINE* to ensure it is properly set up in memory.

The following code defines and initializes a workqueue:

```
#define MY_STACK_SIZE 512
#define MY_PRIORITY 5

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);

struct k_work_q my_work_q;

k_work_queue_init(&my_work_q);

k_work_queue_start(&my_work_q, my_stack_area,
                   K_THREAD_STACK_SIZEOF(my_stack_area), MY_PRIORITY,
                   NULL);
```

In addition the queue identity and certain behavior related to thread rescheduling can be controlled by the optional final parameter; see *k_work_queue_start()* for details.

The following API can be used to interact with a workqueue:

- *k_work_queue_drain()* can be used to block the caller until the work queue has no items left. Work items resubmitted from the workqueue thread are accepted while a queue is draining, but work items from any other thread or ISR are rejected. The restriction on submitting more work can be extended past the completion of the drain operation in order to allow the blocking thread to perform additional work while the queue is "plugged". Note that draining a queue has no effect on scheduling or processing delayable items, but if the queue is plugged and the deadline expires the item will silently fail to be submitted.
- *k_work_queue_unplug()* removes any previous block on submission to the queue due to a previous drain operation.

Submitting a Work Item A work item is defined using a variable of type *k_work*. It must be initialized by calling *k_work_init()*, unless it is defined using *K_WORK_DEFINE* in which case initialization is performed at compile-time.

An initialized work item can be submitted to the system workqueue by calling *k_work_submit()*, or to a specified workqueue by calling *k_work_submit_to_queue()*.

The following code demonstrates how an ISR can offload the printing of error messages to the system workqueue. Note that if the ISR attempts to resubmit the work item while it is still queued, the work item is left unchanged and the associated error message will not be printed.

```
struct device_info {
    struct k_work work;
    char name[16]
} my_device;

void my_isr(void *arg)
{
    ...
    if (error detected) {
        k_work_submit(&my_device.work);
    }
    ...
}

void print_error(struct k_work *item)
{
    struct device_info *the_device =
        CONTAINER_OF(item, struct device_info, work);
    printk("Got error on device %s\n", the_device->name);
}

/* initialize name info for a device */
strcpy(my_device.name, "FOO_dev");

/* initialize work item for printing device's error messages */
k_work_init(&my_device.work, print_error);

/* install my_isr() as interrupt handler for the device (not shown) */
...
```

The following API can be used to check the status of or synchronize with the work item:

- *k_work_busy_get()* returns a snapshot of flags indicating work item state. A zero value indicates the work is not scheduled, submitted, being executed, or otherwise still being referenced by the workqueue infrastructure.
- *k_work_is_pending()* is a helper that indicates `true` if and only if the work is scheduled, queued, or running.
- *k_work_flush()* may be invoked from threads to block until the work item has completed. It returns immediately if the work is not pending.
- *k_work_cancel()* attempts to prevent the work item from being executed. This may or may not be successful. This is safe to invoke from ISRs.
- *k_work_cancel_sync()* may be invoked from threads to block until the work completes; it will return immediately if the cancellation was successful or not necessary (the work wasn't submitted or running). This can be used after *k_work_cancel()* is invoked (from an ISR)` to confirm completion of an ISR-initiated cancellation.

Scheduling a Delayable Work Item A delayable work item is defined using a variable of type *k_work_delayable*. It must be initialized by calling *k_work_init_delayable()*.

For delayed work there are two common use cases, depending on whether a deadline should be extended if a new event occurs. An example is collecting data that comes in asynchronously, e.g. characters from a UART associated with a keyboard. There are two APIs that submit work after a delay:

- *k_work_schedule()* (or *k_work_schedule_for_queue()*) schedules work to be executed at a specific time or after a delay. Further attempts to schedule the same item with this API before the delay completes will not change the time at which the item will be submitted to its queue. Use this if

the policy is to keep collecting data until a specified delay since the **first** unprocessed data was received;

- *k_work_reschedule()* (or *k_work_reschedule_for_queue()*) unconditionally sets the deadline for the work, replacing any previous incomplete delay and changing the destination queue if necessary. Use this if the policy is to keep collecting data until a specified delay since the **last** unprocessed data was received.

If the work item is not scheduled both APIs behave the same. If *K_NO_WAIT* is specified as the delay the behavior is as if the item was immediately submitted directly to the target queue, without waiting for a minimal timeout (unless *k_work_schedule()* is used and a previous delay has not completed).

Both also have variants that allow control of the queue used for submission.

The helper function *k_work_delayable_from_work()* can be used to get a pointer to the containing *k_work_delayable* from a pointer to *k_work* that is passed to a work handler function.

The following additional API can be used to check the status of or synchronize with the work item:

- *k_work_delayable_busy_get()* is the analog to *k_work_busy_get()* for delayable work.
- *k_work_delayable_is_pending()* is the analog to *k_work_is_pending()* for delayable work.
- *k_work_flush_delayable()* is the analog to *k_work_flush()* for delayable work.
- *k_work_cancel_delayable()* is the analog to *k_work_cancel()* for delayable work; similarly with *k_work_cancel_delayable_sync()*.

Synchronizing with Work Items While the state of both regular and delayable work items can be determined from any context using *k_work_busy_get()* and *k_work_delayable_busy_get()* some use cases require synchronizing with work items after they've been submitted. *k_work_flush()*, *k_work_cancel_sync()*, and *k_work_cancel_delayable_sync()* can be invoked from thread context to wait until the requested state has been reached.

These APIs must be provided with a *k_work_sync* object that has no application-inspectable components but is needed to provide the synchronization objects. These objects should not be allocated on a stack if the code is expected to work on architectures with `CONFIG_KERNEL_COHERENCE`.

Workqueue Best Practices

Avoid Race Conditions Sometimes the data a work item must process is naturally thread-safe, for example when it's put into a `k_queue` by some thread and processed in the work thread. More often external synchronization is required to avoid data races: cases where the work thread might inspect or manipulate shared state that's being accessed by another thread or interrupt. Such state might be a flag indicating that work needs to be done, or a shared object that is filled by an ISR or thread and read by the work handler.

For simple flags *Atomic Services* may be sufficient. In other cases spin locks (`k_spinlock_t`) or thread-aware locks (`k_sem`, *k_mutex* , ...) may be used to ensure data races don't occur.

If the selected lock mechanism can *sleep* then allowing the work thread to sleep will starve other work queue items, which may need to make progress in order to get the lock released. Work handlers should try to take the lock with its no-wait path. For example:

```
static void work_handler(struct work *work)
{
        struct work_context *parent = CONTAINER_OF(work, struct work_context,
                                                   work_item);

        if (k_mutex_lock(&parent->lock, K_NO_WAIT) != 0) {
                /* NB: Submit will fail if the work item is being cancelled. */
```

(continues on next page)

(continued from previous page)

```
                (void)k_work_submit(work);
                return;
        }

        /* do stuff under lock */
        k_mutex_unlock(&parent->lock);
        /* do stuff without lock */
}
```

Be aware that if the lock is held by a thread with a lower priority than the work queue the resubmission may starve the thread that would release the lock, causing the application to fail. Where the idiom above is required a delayable work item is preferred, and the work should be (re-)scheduled with a non-zero delay to allow the thread holding the lock to make progress.

Note that submitting from the work handler can fail if the work item had been cancelled. Generally this is acceptable, since the cancellation will complete once the handler finishes. If it is not, the code above must take other steps to notify the application that the work could not be performed.

Work items in isolation are self-locking, so you don't need to hold an external lock just to submit or schedule them. Even if you use external state protected by such a lock to prevent further resubmission, it's safe to do the resubmit as long as you're sure that eventually the item will take its lock and check that state to determine whether it should do anything. Where a delayable work item is being rescheduled in its handler due to inability to take the lock some other self-locking state, such as an atomic flag set by the application/driver when the cancel is initiated, would be required to detect the cancellation and avoid the cancelled work item being submitted again after the deadline.

Check Return Values All work API functions return status of the underlying operation, and in many cases it is important to verify that the intended result was obtained.

- Submitting a work item (*k_work_submit_to_queue()*) can fail if the work is being cancelled or the queue is not accepting new items. If this happens the work will not be executed, which could cause a subsystem that is animated by work handler activity to become non-responsive.
- Asynchronous cancellation (*k_work_cancel()* or *k_work_cancel_delayable()*) can complete while the work item is still being run by a handler. Proceeding to manipulate state shared with the work handler will result in data races that can cause failures.

Many race conditions have been present in Zephyr code because the results of an operation were not checked.

There may be good reason to believe that a return value indicating that the operation did not complete as expected is not a problem. In those cases the code should clearly document this, by (1) casting the return value to `void` to indicate that the result is intentionally ignored, and (2) documenting what happens in the unexpected case. For example:

```
/* If this fails, the work handler will check pub->active and
 * exit without transmitting.
 */
(void)k_work_cancel_delayable(&pub->timer);
```

However in such a case the following code must still avoid data races, as it cannot guarantee that the work thread is not accessing work-related state.

Don't Optimize Prematurely The workqueue API is designed to be safe when invoked from multiple threads and interrupts. Attempts to externally inspect a work item's state and make decisions based on the result are likely to create new problems.

So when new work comes in, just submit it. Don't attempt to "optimize" by checking whether the work item is already submitted by inspecting snapshot state with *k_work_is_pending()* or

k_work_busy_get(), or checking for a non-zero delay from *k_work_delayable_remaining_get()*. Those checks are fragile: a "busy" indication can be obsolete by the time the test is returned, and a "not-busy" indication can also be wrong if work is submitted from multiple contexts, or (for delayable work) if the deadline has completed but the work is still in queued or running state.

A general best practice is to always maintain in shared state some condition that can be checked by the handler to confirm whether there is work to be done. This way you can use the work handler as the standard cleanup path: rather than having to deal with cancellation and cleanup at points where items are submitted, you may be able to have everything done in the work handler itself.

A rare case where you could safely use *k_work_is_pending()* is as a check to avoid invoking *k_work_flush()* or *k_work_cancel_sync()*, if you are *certain* that nothing else might submit the work while you're checking (generally because you're holding a lock that prevents access to state used for submission).

Suggested Uses Use the system workqueue to defer complex interrupt-related processing from an ISR to a shared thread. This allows the interrupt-related processing to be done promptly without compromising the system's ability to respond to subsequent interrupts, and does not require the application to define and manage an additional thread to do the processing.

Configuration Options Related configuration options:

- CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE
- CONFIG_SYSTEM_WORKQUEUE_PRIORITY
- CONFIG_SYSTEM_WORKQUEUE_NO_YIELD

API Reference

group workqueue_apis

Defines

K_WORK_DELAYABLE_DEFINE(work, work_handler)

Initialize a statically-defined delayable work item.

This macro can be used to initialize a statically-defined delayable work item, prior to its first use. For example,

```
static K_WORK_DELAYABLE_DEFINE(<dwork>, <work_handler>);
```

Note that if the runtime dependencies support initialization with *k_work_init_delayable()* using that will eliminate the initialized object in ROM that is produced by this macro and copied in at system startup.

Parameters

- work – Symbol name for delayable work item object
- work_handler – Function to invoke each time work item is processed.

K_WORK_USER_DEFINE(work, work_handler)

Initialize a statically-defined user work item.

This macro can be used to initialize a statically-defined user work item, prior to its first use. For example,

```
static K_WORK_USER_DEFINE(<work>, <work_handler>);
```

Parameters

- `work` – Symbol name for work item object
- `work_handler` – Function to invoke each time work item is processed.

`K_WORK_DEFINE(work, work_handler)`

Initialize a statically-defined work item.

This macro can be used to initialize a statically-defined workqueue work item, prior to its first use. For example,

```
static K_WORK_DEFINE(<work>, <work_handler>);
```

Parameters

- `work` – Symbol name for work item object
- `work_handler` – Function to invoke each time work item is processed.

`K_DELAYED_WORK_DEFINE(work, work_handler)`

Initialize a statically-defined delayed work item.

This macro can be used to initialize a statically-defined workqueue delayed work item, prior to its first use. For example,

```
static K_DELAYED_WORK_DEFINE(<work>, <work_handler>);
```

Parameters

- `work` – Symbol name for delayed work item object
- `work_handler` – Function to invoke each time work item is processed.

Typedefs

typedef void (*`k_work_handler_t`)(struct *k_work* *work)

The signature for a work item handler function.

The function will be invoked by the thread animating a work queue.

Param work the work item that provided the handler.

typedef void (*`k_work_user_handler_t`)(struct k_work_user *work)

Work item handler function type for user work queues.

A work item's handler function is executed by a user workqueue's thread when the work item is processed by the workqueue.

Param work Address of the work item.

Return N/A

Enums

enum `[anonymous]`

Values:

enumerator K_WORK_RUNNING = *BIT*(K_WORK_RUNNING_BIT)

Flag indicating a work item that is running under a work queue thread.

Accessed via *k_work_busy_get()*. May co-occur with other flags.

enumerator K_WORK_CANCELING = *BIT*(K_WORK_CANCELING_BIT)

Flag indicating a work item that is being canceled.

```
Accessed via k_work_busy_get(). May co-occur with other flags.
```

enumerator K_WORK_QUEUED = *BIT*(K_WORK_QUEUED_BIT)

Flag indicating a work item that has been submitted to a queue but has not started running.

Accessed via *k_work_busy_get()*. May co-occur with other flags.

enumerator K_WORK_DELAYED = *BIT*(K_WORK_DELAYED_BIT)

Flag indicating a delayed work item that is scheduled for submission to a queue.

Accessed via *k_work_busy_get()*. May co-occur with other flags.

Functions

void k_work_init(struct *k_work* *work, *k_work_handler_t* handler)

Initialize a (non-delayable) work structure.

This must be invoked before submitting a work structure for the first time. It need not be invoked again on the same work structure. It can be re-invoked to change the associated handler, but this must be done when the work item is idle.

Function properties (list may not be complete) *isr-ok*

Parameters

- work – the work structure to be initialized.
- handler – the handler to be invoked by the work item.

int k_work_busy_get(const struct *k_work* *work)

Busy state flags from the work item.

A zero return value indicates the work item appears to be idle.

Function properties (list may not be complete) *isr-ok*

Note: This is a live snapshot of state, which may change before the result is checked. Use locks where appropriate.

Parameters

- work – pointer to the work item.

Returns a mask of flags K_WORK_DELAYED, K_WORK_QUEUED, K_WORK_RUNNING, and K_WORK_CANCELING.

static inline bool `k_work_is_pending`(const struct *k_work* *work)

Test whether a work item is currently pending.

Wrapper to determine whether a work item is in a non-idle dstate.

Function properties (list may not be complete) *isr-ok*

Note: This is a live snapshot of state, which may change before the result is checked. Use locks where appropriate.

Parameters

- `work` – pointer to the work item.

Returns true if and only if *k_work_busy_get()* returns a non-zero value.

int `k_work_submit_to_queue`(struct *k_work_q* *queue, struct *k_work* *work)

Submit a work item to a queue.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – pointer to the work queue on which the item should run. If NULL the queue from the most recent submission will be used.
- `work` – pointer to the work item.

Return values

- `0` – if work was already submitted to a queue
- `1` – if work was not submitted and has been queued to `queue`
- `2` – if work was running and has been queued to the queue that was running it
- `-EBUSY` –
  - if work submission was rejected because the work item is cancelling; or
  - `queue` is draining; or
  - `queue` is plugged.
- `-EINVAL` – if `queue` is null and the work item has never been run.
- `-ENODEV` – if `queue` has not been started.

int `k_work_submit`(struct *k_work* *work)

Submit a work item to the system queue.

Function properties (list may not be complete) *isr-ok*

Parameters

- `work` – pointer to the work item.

Returns as with *k_work_submit_to_queue()*.

bool `k_work_flush`(struct *k_work* *work, struct *k_work_sync* *sync)

Wait for last-submitted instance to complete.

Resubmissions may occur while waiting, including chained submissions (from within the handler).

Note: Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note: Behavior is undefined if this function is invoked on `work` from a work queue running `work`.

Parameters

- `work` – pointer to the work item.
- `sync` – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- `true` – if call had to wait for completion
- `false` – if work was already idle

int `k_work_cancel`(struct *k_work* *work)

Cancel a work item.

This attempts to prevent a pending (non-delayable) work item from being processed by removing it from the work queue. If the item is being processed, the work item will continue to be processed, but resubmissions are rejected until cancellation completes.

If this returns zero cancellation is complete, otherwise something (probably a work queue thread) is still referencing the item.

See also *k_work_cancel_sync()*.

Function properties (list may not be complete) *isr-ok*

Parameters

- `work` – pointer to the work item.

Returns the *k_work_busy_get()* status indicating the state of the item after all cancellation steps performed by this call are completed.

bool `k_work_cancel_sync`(struct *k_work* *work, struct *k_work_sync* *sync)

Cancel a work item and wait for it to complete.

Same as *k_work_cancel()* but does not return until cancellation is complete. This can be invoked by a thread after *k_work_cancel()* to synchronize with a previous cancellation.

On return the work structure will be idle unless something submits it after the cancellation was complete.

Note: Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note: Behavior is undefined if this function is invoked on `work` from a work queue running `work`.

Parameters

- `work` – pointer to the work item.
- `sync` – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- `true` – if work was pending (call had to wait for cancellation of a running handler to complete, or scheduled or submitted operations were cancelled);
- `false` – otherwise

void `k_work_queue_init`(struct *k_work_q* *queue)

Initialize a work queue structure.

This must be invoked before starting a work queue structure for the first time. It need not be invoked again on the same work queue structure.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – the queue structure to be initialized.

void `k_work_queue_start`(struct *k_work_q* *queue, k_thread_stack_t *stack, size_t stack_size, int prio, const struct *k_work_queue_config* *cfg)

Initialize a work queue.

This configures the work queue thread and starts it running. The function should not be re-invoked on a queue.

Parameters

- `queue` – pointer to the queue structure. It must be initialized in zeroed/bss memory or with *k_work_queue_init* before use.
- `stack` – pointer to the work thread stack area.
- `stack_size` – size of the the work thread stack area, in bytes.
- `prio` – initial thread priority
- `cfg` – optional additional configuration parameters. Pass `NULL` if not required, to use the defaults documented in *k_work_queue_config*.

static inline k_tid_t `k_work_queue_thread_get`(struct *k_work_q* *queue)

Access the thread that animates a work queue.

This is necessary to grant a work queue thread access to things the work items it will process are expected to use.

Parameters

- `queue` – pointer to the queue structure.

Returns the thread associated with the work queue.

int `k_work_queue_drain`(struct *k_work_q* *queue, bool plug)

Wait until the work queue has drained, optionally plugging it.

This blocks submission to the work queue except when coming from queue thread, and blocks the caller until no more work items are available in the queue.

If `plug` is true then submission will continue to be blocked after the drain operation completes until *k_work_queue_unplug()* is invoked.

Note that work items that are delayed are not yet associated with their work queue. They must be cancelled externally if a goal is to ensure the work queue remains empty. The `plug` feature can be used to prevent delayed items from being submitted after the drain completes.

Parameters

- `queue` – pointer to the queue structure.
- `plug` – if true the work queue will continue to block new submissions after all items have drained.

Return values

- `1` – if call had to wait for the drain to complete
- `0` – if call did not have to wait
- `negative` – if wait was interrupted or failed

int `k_work_queue_unplug`(struct *k_work_q* *queue)

Release a work queue to accept new submissions.

This releases the block on new submissions placed when *k_work_queue_drain()* is invoked with the `plug` option enabled. If this is invoked before the drain completes new items may be submitted as soon as the drain completes.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – pointer to the queue structure.

Return values

- `0` – if successfully unplugged
- `-EALREADY` – if the work queue was not plugged.

void `k_work_init_delayable`(struct *k_work_delayable* *dwork, *k_work_handler_t* handler)

Initialize a delayable work structure.

This must be invoked before scheduling a delayable work structure for the first time. It need not be invoked again on the same work structure. It can be re-invoked to change the associated handler, but this must be done when the work item is idle.

Function properties (list may not be complete) *isr-ok*

Parameters

- `dwork` – the delayable work structure to be initialized.
- `handler` – the handler to be invoked by the work item.

static inline struct *k_work_delayable* *`k_work_delayable_from_work`(struct *k_work* *work)

Get the parent delayable work structure from a work pointer.

This function is necessary when a `k_work_handler_t` function is passed to *k_work_schedule_for_queue()* and the handler needs to access data from the container of the containing *`k_work_delayable`*.

Parameters

- `work` – Address passed to the work handler

Returns Address of the containing *`k_work_delayable`* structure.

int `k_work_delayable_busy_get`(const struct *k_work_delayable* *dwork)

Busy state flags from the delayable work item.

Function properties (list may not be complete) *isr-ok*

Note: This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters

- `dwork` – pointer to the delayable work item.

Returns a mask of flags K_WORK_DELAYED, K_WORK_QUEUED, K_WORK_RUNNING, and K_WORK_CANCELING. A zero return value indicates the work item appears to be idle.

static inline bool `k_work_delayable_is_pending`(const struct *k_work_delayable* *dwork)

Test whether a delayed work item is currently pending.

Wrapper to determine whether a delayed work item is in a non-idle state.

Function properties (list may not be complete) *isr-ok*

Note: This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters

- `dwork` – pointer to the delayable work item.

Returns true if and only if *k_work_delayable_busy_get()* returns a non-zero value.

static inline *k_ticks_t* `k_work_delayable_expires_get`(const struct *k_work_delayable* *dwork)

Get the absolute tick count at which a scheduled delayable work will be submitted.

Function properties (list may not be complete) *isr-ok*

Note: This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters

- `dwork` – pointer to the delayable work item.

Returns the tick count when the timer that will schedule the work item will expire, or the current tick count if the work is not scheduled.

static inline *k_ticks_t* `k_work_delayable_remaining_get`(const struct *k_work_delayable* *dwork)

Get the number of ticks until a scheduled delayable work will be submitted.

Function properties (list may not be complete) *isr-ok*

Note: This is a live snapshot of state, which may change before the result can be inspected. Use locks where appropriate.

Parameters

- `dwork` – pointer to the delayable work item.

Returns the number of ticks until the timer that will schedule the work item will expire, or zero if the item is not scheduled.

int `k_work_schedule_for_queue`(struct *k_work_q* *queue, struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Submit an idle work item to a queue after a delay.

Unlike *k_work_reschedule_for_queue()* this is a no-op if the work item is already scheduled or submitted, even if `delay` is `K_NO_WAIT`.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – the queue on which the work item should be submitted after the delay.
- `dwork` – pointer to the delayable work item.
- `delay` – the time to wait before submitting the work item. If `K_NO_WAIT` and the work is not pending this is equivalent to *k_work_submit_to_queue()*.

Return values

- `0` – if work was already scheduled or submitted.
- `1` – if work has been scheduled.
- `-EBUSY` – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.
- `-EINVAL` – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.
- `-ENODEV` – if `delay` is `K_NO_WAIT` and *k_work_submit_to_queue()* fails with this code.

int k_work_schedule(struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Submit an idle work item to the system work queue after a delay.

This is a thin wrapper around *k_work_schedule_for_queue()*, with all the API characteristcs of that function.

Parameters

- dwork – pointer to the delayable work item.
- delay – the time to wait before submitting the work item. If K_NO_WAIT this is equivalent to *k_work_submit_to_queue()*.

Returns as with *k_work_schedule_for_queue()*.

int k_work_reschedule_for_queue(struct *k_work_q* *queue, struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Reschedule a work item to a queue after a delay.

Unlike *k_work_schedule_for_queue()* this function can change the deadline of a scheduled work item, and will schedule a work item that isn't idle (e.g. is submitted or running). This function does not affect ("unsubmit") a work item that has been submitted to a queue.

Function properties (list may not be complete) *isr-ok*

Note: If delay is K_NO_WAIT ("no delay") the return values are as with *k_work_submit_to_queue()*.

Parameters

- queue – the queue on which the work item should be submitted after the delay.
- dwork – pointer to the delayable work item.
- delay – the time to wait before submitting the work item. If K_NO_WAIT this is equivalent to *k_work_submit_to_queue()* after canceling any previous scheduled submission.

Return values

- 0 – if delay is K_NO_WAIT and work was already on a queue
- 1 – if
  - delay is K_NO_WAIT and work was not submitted but has now been queued to queue; or
  - delay not K_NO_WAIT and work has been scheduled
- 2 – if delay is K_NO_WAIT and work was running and has been queued to the queue that was running it
- -EBUSY – if delay is K_NO_WAIT and *k_work_submit_to_queue()* fails with this code.
- -EINVAL – if delay is K_NO_WAIT and *k_work_submit_to_queue()* fails with this code.
- -ENODEV – if delay is K_NO_WAIT and *k_work_submit_to_queue()* fails with this code.

int `k_work_reschedule`(struct *k_work_delayable* *dwork, *k_timeout_t* delay)

Reschedule a work item to the system work queue after a delay.

This is a thin wrapper around *k_work_reschedule_for_queue()*, with all the API characteristcs of that function.

Parameters

- `dwork` – pointer to the delayable work item.
- `delay` – the time to wait before submitting the work item.

Returns as with *k_work_reschedule_for_queue()*.

bool `k_work_flush_delayable`(struct *k_work_delayable* *dwork, struct *k_work_sync* *sync)

Flush delayable work.

If the work is scheduled, it is immediately submitted. Then the caller blocks until the work completes, as with *k_work_flush()*.

Note: Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note: Behavior is undefined if this function is invoked on `dwork` from a work queue running `dwork`.

Parameters

- `dwork` – pointer to the delayable work item.
- `sync` – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- `true` – if call had to wait for completion
- `false` – if work was already idle

int `k_work_cancel_delayable`(struct *k_work_delayable* *dwork)

Cancel delayable work.

Similar to *k_work_cancel()* but for delayable work. If the work is scheduled or submitted it is canceled. This function does not wait for the cancellation to complete.

Function properties (list may not be complete) *isr-ok*

Note: The work may still be running when this returns. Use *k_work_flush_delayable()* or *k_work_cancel_delayable_sync()* to ensure it is not running.

Note: Canceling delayable work does not prevent rescheduling it. It does prevent submitting it until the cancellation completes.

Parameters

- `dwork` – pointer to the delayable work item.

Returns the *k_work_delayable_busy_get()* status indicating the state of the item after all cancellation steps performed by this call are completed.

bool `k_work_cancel_delayable_sync`(struct *k_work_delayable* *dwork, struct *k_work_sync* *sync)

Cancel delayable work and wait.

Like *k_work_cancel_delayable()* but waits until the work becomes idle.

Note: Canceling delayable work does not prevent rescheduling it. It does prevent submitting it until the cancellation completes.

Note: Be careful of caller and work queue thread relative priority. If this function sleeps it will not return until the work queue thread completes the tasks that allow this thread to resume.

Note: Behavior is undefined if this function is invoked on `dwork` from a work queue running `dwork`.

Parameters

- `dwork` – pointer to the delayable work item.
- `sync` – pointer to an opaque item containing state related to the pending cancellation. The object must persist until the call returns, and be accessible from both the caller thread and the work queue thread. The object must not be used for any other flush or cancel operation until this one completes. On architectures with CONFIG_KERNEL_COHERENCE the object must be allocated in coherent memory.

Return values

- `true` – if work was not idle (call had to wait for cancellation of a running handler to complete, or scheduled or submitted operations were cancelled);
- `false` – otherwise

static inline bool `k_work_pending`(const struct *k_work* *work)

static inline void `k_work_q_start`(struct *k_work_q* *work_q, k_thread_stack_t *stack, size_t stack_size, int prio)

static inline void `k_delayed_work_init`(struct *k_delayed_work* *work, *k_work_handler_t* handler)

static inline int `k_delayed_work_submit_to_queue`(struct *k_work_q* *work_q, struct *k_delayed_work* *work, *k_timeout_t* delay)

static inline int `k_delayed_work_submit`(struct *k_delayed_work* *work, *k_timeout_t* delay)

static inline int `k_delayed_work_cancel`(struct *k_delayed_work* *work)

static inline bool `k_delayed_work_pending`(struct *k_delayed_work* *work)

static inline int32_t k_delayed_work_remaining_get(struct *k_delayed_work* *work)

static inline *k_ticks_t* k_delayed_work_expires_ticks(struct *k_delayed_work* *work)

static inline *k_ticks_t* k_delayed_work_remaining_ticks(struct *k_delayed_work* *work)

static inline void k_work_user_init(struct k_work_user *work, *k_work_user_handler_t* handler)

Initialize a userspace work item.

This routine initializes a user workqueue work item, prior to its first use.

Parameters

- work – Address of work item.
- handler – Function to invoke each time work item is processed.

Returns N/A

static inline bool k_work_user_is_pending(struct k_work_user *work)

Check if a userspace work item is pending.

This routine indicates if user work item *work* is pending in a workqueue's queue.

Function properties (list may not be complete) *isr-ok*

Note: Checking if the work is pending gives no guarantee that the work will still be pending when this information is used. It is up to the caller to make sure that this information is used in a safe manner.

Parameters

- work – Address of work item.

Returns true if work item is pending, or false if it is not pending.

static inline int k_work_user_submit_to_queue(struct k_work_user_q *work_q, struct k_work_user *work)

Submit a work item to a user mode workqueue.

Submits a work item to a workqueue that runs in user mode. A temporary memory allocation is made from the caller's resource pool which is freed once the worker thread consumes the *k_work* item. The workqueue thread must have memory access to the *k_work* item being submitted. The caller must have permission granted on the work_q parameter's queue object.

Function properties (list may not be complete) *isr-ok*

Parameters

- work_q – Address of workqueue.
- work – Address of work item.

Return values

- -EBUSY – if the work item was already in some workqueue
- -ENOMEM – if no memory for thread resource pool allocation
- 0 – Success

void k_work_user_queue_start(struct k_work_user_q *work_q, k_thread_stack_t *stack, size_t stack_size, int prio, const char *name)

Start a workqueue in user mode.

This works identically to *k_work_queue_start()* except it is callable from user mode, and the worker thread created will run in user mode. The caller must have permissions granted on both the work_q parameter's thread and queue objects, and the same restrictions on priority apply as *k_thread_create()*.

Parameters

- `work_q` – Address of workqueue.
- `stack` – Pointer to work queue thread's stack space, as defined by *K_THREAD_STACK_DEFINE()*
- `stack_size` – Size of the work queue thread's stack (in bytes), which should either be the same constant passed to *K_THREAD_STACK_DEFINE()* or the value of *K_THREAD_STACK_SIZEOF()*.
- `prio` – Priority of the work queue's thread.
- `name` – optional thread name. If not null a copy is made into the thread's name buffer.

Returns N/A

void k_work_poll_init(struct k_work_poll *work, *k_work_handler_t* handler)

Initialize a triggered work item.

This routine initializes a workqueue triggered work item, prior to its first use.

Parameters

- `work` – Address of triggered work item.
- `handler` – Function to invoke each time work item is processed.

Returns N/A

int k_work_poll_submit_to_queue(struct *k_work_q* *work_q, struct k_work_poll *work, struct *k_poll_event* *events, int num_events, *k_timeout_t* timeout)

Submit a triggered work item.

This routine schedules work item *work* to be processed by workqueue *work_q* when one of the given *events* is signaled. The routine initiates internal poller for the work item and then returns to the caller. Only when one of the watched events happen the work item is actually submitted to the workqueue and becomes pending.

Submitting a previously submitted triggered work item that is still waiting for the event cancels the existing submission and reschedules it the using the new event list. Note that this behavior is inherently subject to race conditions with the pre-existing triggered work item and work queue, so care must be taken to synchronize such resubmissions externally.

Function properties (list may not be complete) *isr-ok*

Warning: Provided array of events as well as a triggered work item must be placed in persistent memory (valid until work handler execution or work cancellation) and cannot be modified after submission.

Parameters

- `work_q` – Address of workqueue.

- `work` – Address of delayed work item.
- `events` – An array of events which trigger the work.
- `num_events` – The number of events in the array.
- `timeout` – Timeout after which the work will be scheduled for execution even if not triggered.

Return values

- `0` – Work item started watching for events.
- `-EINVAL` – Work item is being processed or has completed its work.
- `-EADDRINUSE` – Work item is pending on a different workqueue.

int `k_work_poll_submit`(struct k_work_poll *work, struct *k_poll_event* *events, int num_events, *k_timeout_t* timeout)

Submit a triggered work item to the system workqueue.

This routine schedules work item *work* to be processed by system workqueue when one of the given *events* is signaled. The routine initiates internal poller for the work item and then returns to the caller. Only when one of the watched events happen the work item is actually submitted to the workqueue and becomes pending.

Submitting a previously submitted triggered work item that is still waiting for the event cancels the existing submission and reschedules it the using the new event list. Note that this behavior is inherently subject to race conditions with the pre-existing triggered work item and work queue, so care must be taken to synchronize such resubmissions externally.

Function properties (list may not be complete) *isr-ok*

Warning: Provided array of events as well as a triggered work item must not be modified until the item has been processed by the workqueue.

Parameters

- `work` – Address of delayed work item.
- `events` – An array of events which trigger the work.
- `num_events` – The number of events in the array.
- `timeout` – Timeout after which the work will be scheduled for execution even if not triggered.

Return values

- `0` – Work item started watching for events.
- `-EINVAL` – Work item is being processed or has completed its work.
- `-EADDRINUSE` – Work item is pending on a different workqueue.

int `k_work_poll_cancel`(struct k_work_poll *work)

Cancel a triggered work item.

This routine cancels the submission of triggered work item *work*. A triggered work item can only be canceled if no event triggered work submission.

Function properties (list may not be complete) *isr-ok*

Parameters

- `work` – Address of delayed work item.

Return values

- `0` – Work item canceled.
- `-EINVAL` – Work item is being processed or has completed its work.

struct `k_work`

#include <kernel.h> A structure used to submit work.

struct `k_work_delayable`

#include <kernel.h> A structure used to submit work after a delay.

struct `k_work_sync`

#include <kernel.h> A structure holding internal state for a pending synchronous operation on a work item or queue.

Instances of this type are provided by the caller for invocation of *k_work_flush()*, *k_work_cancel_sync()* and sibling flush and cancel APIs. A referenced object must persist until the call returns, and be accessible from both the caller thread and the work queue thread.

Note: If CONFIG_KERNEL_COHERENCE is enabled the object must be allocated in coherent memory; see *arch_mem_coherent()*. The stack on these architectures is generally not coherent. be stack-allocated. Violations are detected by runtime assertion.

---

struct `k_work_queue_config`

#include <kernel.h> A structure holding optional configuration items for a work queue.

This structure, and values it references, are not retained by *k_work_queue_start()*.

Public Members

const char *`name`

The name to be given to the work queue thread.

If left null the thread will not have a name.

bool `no_yield`

Control whether the work queue thread should yield between items.

Yielding between items helps guarantee the work queue thread does not starve other threads, including cooperative ones released by a work item. This is the default behavior.

Set this to `true` to prevent the work queue thread from yielding between items. This may be appropriate when a sequence of items should complete without yielding control.

struct `k_work_q`

#include <kernel.h> A structure used to hold work until it can be processed.

struct `k_delayed_work`

#include <kernel.h>

Zephyr Without Threads

Thread support is not necessary in some applications:

- Bootloaders
- Simple event-driven applications
- Examples intended to demonstrate core functionality

Thread support can be disabled in Zephyr by setting `CONFIG_MULTITHREADING` to `n`. Since this configuration has a significant impact on Zephyr's functionality and testing of it has been limited, there are conditions on what can be expected to work in this configuration.

What Can be Expected to Work These core capabilities shall function correctly when `CONFIG_MULTITHREADING` is disabled:

- The *build system*
- The ability to boot the application to `main()`
- *Interrupt management*
- The system clock including *`k_uptime_get()`*
- Timers, i.e. `k_timer()`
- Non-sleeping delays e.g. *`k_busy_wait()`*.
- Sleeping *`k_cpu_idle()`*.
- Pre `main()` drivers and subsystems initialization e.g. *`SYS_INIT`*.
- *Memory Management*
- Specifically identified drivers in certain subsystems, listed below.

The expectations above affect selection of other features; for example `CONFIG_SYS_CLOCK_EXISTS` cannot be set to `n`.

What Cannot be Expected to Work Functionality that will not work with `CONFIG_MULTITHREADING` includes majority of the kernel API:

- *Threads*
- *Scheduling*
- *Workqueue Threads*
- *Polling API*
- *Semaphores*
- *Mutexes*
- *Condition Variables*
- *Data Passing*

Subsystem Behavior Without Thread Support The sections below list driver and functional subsystems that are expected to work to some degree when `CONFIG_MULTITHREADING` is disabled. Subsystems that are not listed here should not be expected to work.

Some existing drivers within the listed subsystems do not work when threading is disabled, but are within scope based on their subsystem, or may be sufficiently isolated that supporting them on a particular platform is low-impact. Enhancements to add support to existing capabilities that were not originally implemented to work with threads disabled will be considered.

Flash The *Flash* is expected to work for all SoC flash peripheral drivers. Bus-accessed devices like serial memories may not be supported.

List/table of supported drivers to go here

GPIO The *GPIO* is expected to work for all SoC GPIO peripheral drivers. Bus-accessed devices like GPIO extenders may not be supported.

List/table of supported drivers to go here

UART A subset of the *UART* is expected to work for all SoC UART peripheral drivers.

- Applications that select `CONFIG_UART_INTERRUPT_DRIVEN` may work, depending on driver implementation.
- Applications that select `CONFIG_UART_ASYNC_API` may work, depending on driver implementation.
- Applications that do not select either `CONFIG_UART_ASYNC_API` or `CONFIG_UART_INTERRUPT_DRIVEN` are expected to work.

List/table of supported drivers to go here, including which API options are supported

Interrupts

An *interrupt service routine* (ISR) is a function that executes asynchronously in response to a hardware or software interrupt. An ISR normally preempts the execution of the current thread, allowing the response to occur with very low overhead. Thread execution resumes only once all ISR work has been completed.

Concepts Any number of ISRs can be defined (limited only by available RAM), subject to the constraints imposed by underlying hardware.

An ISR has the following key properties:

- An **interrupt request (IRQ) signal** that triggers the ISR.
- A **priority level** associated with the IRQ.
- An **interrupt handler function** that is invoked to handle the interrupt.
- An **argument value** that is passed to that function.

An IDT (Interrupt Descriptor Table) or a vector table is used to associate a given interrupt source with a given ISR. Only a single ISR can be associated with a specific IRQ at any given time.

Multiple ISRs can utilize the same function to process interrupts, allowing a single function to service a device that generates multiple types of interrupts or to service multiple devices (usually of the same type). The argument value passed to an ISR's function allows the function to determine which interrupt has been signaled.

The kernel provides a default ISR for all unused IDT entries. This ISR generates a fatal system error if an unexpected interrupt is signaled.

The kernel supports **interrupt nesting**. This allows an ISR to be preempted in mid-execution if a higher priority interrupt is signaled. The lower priority ISR resumes execution once the higher priority ISR has completed its processing.

An ISR's interrupt handler function executes in the kernel's **interrupt context**. This context has its own dedicated stack area (or, on some architectures, stack areas). The size of the interrupt context stack must be capable of handling the execution of multiple concurrent ISRs if interrupt nesting support is enabled.

Important: Many kernel APIs can be used only by threads, and not by ISRs. In cases where a routine may be invoked by both threads and ISRs the kernel provides the *k_is_in_isr()* API to allow the routine to alter its behavior depending on whether it is executing as part of a thread or as part of an ISR.

Multi-level Interrupt handling A hardware platform can support more interrupt lines than natively-provided through the use of one or more nested interrupt controllers. Sources of hardware interrupts are combined into one line that is then routed to the parent controller.

If nested interrupt controllers are supported, `CONFIG_MULTI_LEVEL_INTERRUPTS` should be set to 1, and `CONFIG_2ND_LEVEL_INTERRUPTS` and `CONFIG_3RD_LEVEL_INTERRUPTS` configured as well, based on the hardware architecture.

A unique 32-bit interrupt number is assigned with information embedded in it to select and invoke the correct Interrupt Service Routine (ISR). Each interrupt level is given a byte within this 32-bit number, providing support for up to four interrupt levels using this arch, as illustrated and explained below:

```
          9             2   0
    - - - - - - - - - - - - -          (LEVEL 1)
  5       |         A   |
- - - - - - -           - - - - - - -  (LEVEL 2)
  |   C                       B
- - - - - - -                          (LEVEL 3)
        D
```

There are three interrupt levels shown here.

- '-' means interrupt line and is numbered from 0 (right most).
- LEVEL 1 has 12 interrupt lines, with two lines (2 and 9) connected to nested controllers and one device 'A' on line 4.
- One of the LEVEL 2 controllers has interrupt line 5 connected to a LEVEL 3 nested controller and one device 'C' on line 3.
- The other LEVEL 2 controller has no nested controllers but has one device 'B' on line 2.
- The LEVEL 3 controller has one device 'D' on line 2.

Here's how unique interrupt numbers are generated for each hardware interrupt. Let's consider four interrupts shown above as A, B, C, and D:

```
A -> 0x00000004
B -> 0x00000302
C -> 0x00000409
D -> 0x00030609
```

Note: The bit positions for LEVEL 2 and onward are offset by 1, as 0 means that interrupt number is not present for that level. For our example, the LEVEL 3 controller has device D on line 2, connected to the LEVEL 2 controller's line 5, that is connected to the LEVEL 1 controller's line 9 (2 -> 5 -> 9). Because of the encoding offset for LEVEL 2 and onward, device D is given the number 0x00030609.

Preventing Interruptions In certain situations it may be necessary for the current thread to prevent ISRs from executing while it is performing time-sensitive or critical section operations.

A thread may temporarily prevent all IRQ handling in the system using an **IRQ lock**. This lock can be applied even when it is already in effect, so routines can use it without having to know if it is already in effect. The thread must unlock its IRQ lock the same number of times it was locked before interrupts can be once again processed by the kernel while the thread is running.

Important: The IRQ lock is thread-specific. If thread A locks out interrupts then performs an operation that puts itself to sleep (e.g. sleeping for N milliseconds), the thread's IRQ lock no longer applies once thread A is swapped out and the next ready thread B starts to run.

This means that interrupts can be processed while thread B is running unless thread B has also locked out interrupts using its own IRQ lock. (Whether interrupts can be processed while the kernel is switching between two threads that are using the IRQ lock is architecture-specific.)

When thread A eventually becomes the current thread once again, the kernel re-establishes thread A's IRQ lock. This ensures thread A won't be interrupted until it has explicitly unlocked its IRQ lock.

If thread A does not sleep but does make a higher-priority thread B ready, the IRQ lock will inhibit any preemption that would otherwise occur. Thread B will not run until the next *reschedule point* reached after releasing the IRQ lock.

Alternatively, a thread may temporarily **disable** a specified IRQ so its associated ISR does not execute when the IRQ is signaled. The IRQ must be subsequently **enabled** to permit the ISR to execute.

Important: Disabling an IRQ prevents *all* threads in the system from being preempted by the associated ISR, not just the thread that disabled the IRQ.

Zero Latency Interrupts Preventing interruptions by applying an IRQ lock may increase the observed interrupt latency. A high interrupt latency, however, may not be acceptable for certain low-latency use-cases.

The kernel addresses such use-cases by allowing interrupts with critical latency constraints to execute at a priority level that cannot be blocked by interrupt locking. These interrupts are defined as *zero-latency interrupts*. The support for zero-latency interrupts requires `CONFIG_ZERO_LATENCY_IRQS` to be enabled. In addition to that, the flag `IRQ_ZERO_LATENCY` must be passed to *`IRQ_CONNECT`* or *`IRQ_DIRECT_CONNECT`* macros to configure the particular interrupt with zero latency.

Zero-latency interrupts are expected to be used to manage hardware events directly, and not to inter-operate with the kernel code at all. They should treat all kernel APIs as undefined behavior (i.e. an application that uses the APIs inside a zero-latency interrupt context is responsible for directly verifying correct behavior). Zero-latency interrupts may not modify any data inspected by kernel APIs invoked from normal Zephyr contexts and shall not generate exceptions that need to be handled synchronously (e.g. kernel panic).

Important: Zero-latency interrupts are supported on an architecture-specific basis. The feature is currently implemented in the ARM Cortex-M architecture variant.

Offloading ISR Work An ISR should execute quickly to ensure predictable system operation. If time consuming processing is required the ISR should offload some or all processing to a thread, thereby restoring the kernel's ability to respond to other interrupts.

The kernel supports several mechanisms for offloading interrupt-related processing to a thread.

- An ISR can signal a helper thread to do interrupt-related processing using a kernel object, such as a FIFO, LIFO, or semaphore.
- An ISR can instruct the system workqueue thread to execute a work item. (See *Workqueue Threads*.)

When an ISR offloads work to a thread, there is typically a single context switch to that thread when the ISR completes, allowing interrupt-related processing to continue almost immediately. However, depending on the priority of the thread handling the offload, it is possible that the currently executing cooperative thread or other higher-priority threads may execute before the thread handling the offload is scheduled.

Implementation

Defining a regular ISR An ISR is defined at runtime by calling *IRQ_CONNECT*. It must then be enabled by calling *irq_enable()*.

Important: IRQ_CONNECT() is not a C function and does some inline assembly magic behind the scenes. All its arguments must be known at build time. Drivers that have multiple instances may need to define per-instance config functions to configure each instance of the interrupt.

The following code defines and enables an ISR.

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_ISR_ARG  DEVICE_GET(my_device)
#define MY_IRQ_FLAGS 0       /* IRQ flags */

void my_isr(void *arg)
{
   ... /* ISR code */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Since the *IRQ_CONNECT* macro requires that all its parameters be known at build time, in some cases this may not be acceptable. It is also possible to install interrupts at runtime with *irq_connect_dynamic()*. It is used in exactly the same way as *IRQ_CONNECT*:

```
void my_isr_installer(void)
{
   ...
   irq_connect_dynamic(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG,
                       MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Dynamic interrupts require the CONFIG_DYNAMIC_INTERRUPTS option to be enabled. Removing or re-configuring a dynamic interrupt is currently unsupported.

Defining a 'direct' ISR Regular Zephyr interrupts introduce some overhead which may be unacceptable for some low-latency use-cases. Specifically:

- The argument to the ISR is retrieved and passed to the ISR
- If power management is enabled and the system was idle, all the hardware will be resumed from low-power state before the ISR is executed, which can be very time-consuming
- Although some architectures will do this in hardware, other architectures need to switch to the interrupt stack in code
- After the interrupt is serviced, the OS then performs some logic to potentially make a scheduling decision.

Zephyr supports so-called 'direct' interrupts, which are installed via *IRQ_DIRECT_CONNECT*. These direct interrupts have some special implementation requirements and a reduced feature set; see the definition of *IRQ_DIRECT_CONNECT* for details.

The following code demonstrates a direct ISR:

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_IRQ_FLAGS 0       /* IRQ flags */

ISR_DIRECT_DECLARE(my_isr)
{
   do_stuff();
   ISR_DIRECT_PM(); /* PM done after servicing interrupt for best latency */
   return 1; /* We should check if scheduling decision should be made */
}

void my_isr_installer(void)
{
   ...
   IRQ_DIRECT_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Installation of dynamic direct interrupts is supported on an architecture-specific basis. (The feature is currently implemented in ARM Cortex-M architecture variant. Dynamic direct interrupts feature is exposed to the user via an ARM-only API.)

Implementation Details Interrupt tables are set up at build time using some special build tools. The details laid out here apply to all architectures except x86, which are covered in the *x86 Details* section below.

Any invocation of *IRQ_CONNECT* will declare an instance of struct _isr_list which is placed in a special .intList section:

```
struct _isr_list {
    /** IRQ line number */
    int32_t irq;
    /** Flags for this IRQ, see ISR_FLAG_* definitions */
    int32_t flags;
    /** ISR to call */
    void *func;
    /** Parameter for non-direct IRQs */
    void *param;
};
```

Zephyr is built in two phases; the first phase of the build produces ${ZEPHYR_PREBUILT_EXECUTABLE}.elf which contains all the entries in the .intList section preceded by a header:

```
struct {
    void *spurious_irq_handler;
    void *sw_irq_handler;
    uint32_t num_isrs;
    uint32_t num_vectors;
    struct _isr_list isrs[];  <- of size num_isrs
};
```

This data consisting of the header and instances of struct _isr_list inside ${ZEPHYR_PREBUILT_EXECUTABLE}.elf is then used by the gen_isr_tables.py script to generate a C file defining a vector table and software ISR table that are then compiled and linked into the final application.

The priority level of any interrupt is not encoded in these tables, instead *IRQ_CONNECT* also has a runtime component which programs the desired priority level of the interrupt to the interrupt controller. Some architectures do not support the notion of interrupt priority, in which case the priority argument is ignored.

Vector Table A vector table is generated when CONFIG_GEN_IRQ_VECTOR_TABLE is enabled. This data structure is used natively by the CPU and is simply an array of function pointers, where each element n corresponds to the IRQ handler for IRQ line n, and the function pointers are:

1. For 'direct' interrupts declared with *IRQ_DIRECT_CONNECT*, the handler function will be placed here.
2. For regular interrupts declared with *IRQ_CONNECT*, the address of the common software IRQ handler is placed here. This code does common kernel interrupt bookkeeping and looks up the ISR and parameter from the software ISR table.
3. For interrupt lines that are not configured at all, the address of the spurious IRQ handler will be placed here. The spurious IRQ handler causes a system fatal error if encountered.

Some architectures (such as the Nios II internal interrupt controller) have a common entry point for all interrupts and do not support a vector table, in which case the CONFIG_GEN_IRQ_VECTOR_TABLE option should be disabled.

Some architectures may reserve some initial vectors for system exceptions and declare this in a table elsewhere, in which case CONFIG_GEN_IRQ_START_VECTOR needs to be set to properly offset the indices in the table.

SW ISR Table This is an array of struct _isr_table_entry:

```
struct _isr_table_entry {
    void *arg;
    void (*isr)(void *);
};
```

This is used by the common software IRQ handler to look up the ISR and its argument and execute it. The active IRQ line is looked up in an interrupt controller register and used to index this table.

x86 Details The x86 architecture has a special type of vector table called the Interrupt Descriptor Table (IDT) which must be laid out in a certain way per the x86 processor documentation. It is still fundamentally a vector table, and the *arch/x86/gen_idt.py* tool uses the .intList section to create it. However, on APIC-based systems the indexes in the vector table do not correspond to the IRQ line. The first 32 vectors are reserved for CPU exceptions, and all remaining vectors (up to index 255) correspond to the priority level, in groups of 16. In this scheme, interrupts of priority level 0 will be placed in vectors 32-47, level 1 48-63, and so forth. When the *arch/x86/gen_idt.py* tool is constructing the IDT, when it configures an interrupt it will look for a free vector in the appropriate range for the requested priority level and set the handler there.

On x86 when an interrupt or exception vector is executed by the CPU, there is no foolproof way to determine which vector was fired, so a software ISR table indexed by IRQ line is not used. Instead, the *IRQ_CONNECT* call creates a small assembly language function which calls the common interrupt code in `_interrupt_enter()` with the ISR and parameter as arguments. It is the address of this assembly interrupt stub which gets placed in the IDT. For interrupts declared with *IRQ_DIRECT_CONNECT* the parameterless ISR is placed directly in the IDT.

On systems where the position in the vector table corresponds to the interrupt's priority level, the interrupt controller needs to know at runtime what vector is associated with an IRQ line. *arch/x86/gen_idt.py* additionally creates an _irq_to_interrupt_vector array which maps an IRQ line to its configured vector in the IDT. This is used at runtime by *IRQ_CONNECT* to program the IRQ-to-vector association in the interrupt controller.

For dynamic interrupts, the build must generate some 4-byte dynamic interrupt stubs, one stub per dynamic interrupt in use. The number of stubs is controlled by the `CONFIG_X86_DYNAMIC_IRQ_STUBS` option. Each stub pushes an unique identifier which is then used to fetch the appropriate handler function and parameter out of a table populated when the dynamic interrupt was connected.

Suggested Uses Use a regular or direct ISR to perform interrupt processing that requires a very rapid response, and can be done quickly without blocking.

Note: Interrupt processing that is time consuming, or involves blocking, should be handed off to a thread. See *Offloading ISR Work* for a description of various techniques that can be used in an application.

Configuration Options Related configuration options:

- `CONFIG_ISR_STACK_SIZE`

Additional architecture-specific and device-specific configuration options also exist.

API Reference

group `isr_apis`

Defines

IRQ_CONNECT(irq_p, priority_p, isr_p, isr_param_p, flags_p)

Initialize an interrupt handler.

This routine initializes an interrupt handler for an IRQ. The IRQ must be subsequently enabled before the interrupt handler begins servicing interrupts.

Warning: Although this routine is invoked at run-time, all of its arguments must be computable by the compiler at build time.

Parameters

- irq_p – IRQ line number.
- priority_p – Interrupt priority.
- isr_p – Address of interrupt service routine.
- isr_param_p – Parameter passed to interrupt service routine.
- flags_p – Architecture-specific IRQ configuration flags..

IRQ_DIRECT_CONNECT(irq_p, priority_p, isr_p, flags_p)

Initialize a 'direct' interrupt handler.

This routine initializes an interrupt handler for an IRQ. The IRQ must be subsequently enabled via *irq_enable()* before the interrupt handler begins servicing interrupts.

These ISRs are designed for performance-critical interrupt handling and do not go through common interrupt handling code. They must be implemented in such a way that it is safe to put them directly in the vector table. For ISRs written in C, The *ISR_DIRECT_DECLARE()* macro will do this automatically. For ISRs written in assembly it is entirely up to the developer to ensure that the right steps are taken.

This type of interrupt currently has a few limitations compared to normal Zephyr interrupts:

- No parameters are passed to the ISR.
- No stack switch is done, the ISR will run on the interrupted context's stack, unless the architecture automatically does the stack switch in HW.
- Interrupt locking state is unchanged from how the HW sets it when the ISR runs. On arches that enter ISRs with interrupts locked, they will remain locked.
- Scheduling decisions are now optional, controlled by the return value of ISRs implemented with the *ISR_DIRECT_DECLARE()* macro
- The call into the OS to exit power management idle state is now optional. Normal interrupts always do this before the ISR is run, but when it runs is now controlled by the placement of a *ISR_DIRECT_PM()* macro, or omitted entirely.

Warning: Although this routine is invoked at run-time, all of its arguments must be computable by the compiler at build time.

Parameters

- irq_p – IRQ line number.
- priority_p – Interrupt priority.
- isr_p – Address of interrupt service routine.
- flags_p – Architecture-specific IRQ configuration flags.

ISR_DIRECT_HEADER()

Common tasks before executing the body of an ISR.

This macro must be at the beginning of all direct interrupts and performs minimal architecture-specific tasks before the ISR itself can run. It takes no arguments and has no return value.

ISR_DIRECT_FOOTER(check_reschedule)

Common tasks before exiting the body of an ISR.

This macro must be at the end of all direct interrupts and performs minimal architecture-specific tasks like EOI. It has no return value.

In a normal interrupt, a check is done at end of interrupt to invoke z_swap() logic if the current thread is preemptible and there is another thread ready to run in the kernel's ready queue cache. This is now optional and controlled by the check_reschedule argument. If unsure, set to nonzero. On systems that do stack switching and nested interrupt tracking in software, z_swap() should only be called if this was a non-nested interrupt.

Parameters

- check_reschedule – If nonzero, additionally invoke scheduling logic

ISR_DIRECT_PM()

Perform power management idle exit logic.

This macro may optionally be invoked somewhere in between IRQ_DIRECT_HEADER() and IRQ_DIRECT_FOOTER() invocations. It performs tasks necessary to exit power management idle state. It takes no parameters and returns no arguments. It may be omitted, but be careful!

ISR_DIRECT_DECLARE(name)

Helper macro to declare a direct interrupt service routine.

This will declare the function in a proper way and automatically include the *ISR_DIRECT_FOOTER()* and *ISR_DIRECT_HEADER()* macros. The function should return nonzero status if a scheduling decision should potentially be made. See *ISR_DIRECT_FOOTER()* for more details on the scheduling decision.

For architectures that support 'regular' and 'fast' interrupt types, where these interrupt types require different assembly language handling of registers by the ISR, this will always generate code for the 'fast' interrupt type.

Example usage:

```
ISR_DIRECT_DECLARE(my_isr)
{
        bool done = do_stuff();
        ISR_DIRECT_PM(); // done after do_stuff() due to latency concerns
        if (!done) {
            return 0; // don't bother checking if we have to z_swap()
        }

        k_sem_give(some_sem);
        return 1;
 }
```

Parameters

- name – symbol name of the ISR

irq_lock()

Lock interrupts.

This routine disables all interrupts on the CPU. It returns an unsigned integer "lock-out key", which is an architecture-dependent indicator of whether interrupts were locked prior to the call. The lock-out key must be passed to *irq_unlock()* to re-enable interrupts.

This routine can be called recursively, as long as the caller keeps track of each lock-out key that is generated. Interrupts are re-enabled by passing each of the keys to *irq_unlock()* in the reverse order they were acquired. (That is, each call to *irq_lock()* must be balanced by a corresponding call to *irq_unlock()*.)

This routine can only be invoked from supervisor mode. Some architectures (for example, ARM) will fail silently if invoked from user mode instead of generating an exception.

Note: This routine must also serve as a memory barrier to ensure the uniprocessor implementation of `k_spinlock_t` is correct.

Note: This routine can be called by ISRs or by threads. If it is called by a thread, the interrupt lock is thread-specific; this means that interrupts remain disabled only while the thread is running. If the thread performs an operation that allows another thread to run (for example, giving a semaphore or sleeping for N milliseconds), the interrupt lock no longer applies and interrupts may be re-enabled while other processing occurs. When the thread once again becomes the current thread, the kernel re-establishes its interrupt lock; this ensures the thread won't be interrupted until it has explicitly released the interrupt lock it established.

Warning: The lock-out key should never be used to manually re-enable interrupts or to inspect or manipulate the contents of the CPU's interrupt bits.

Returns An architecture-dependent lock-out key representing the "interrupt disable state" prior to the call.

`irq_unlock(key)`

Unlock interrupts.

This routine reverses the effect of a previous call to *irq_lock()* using the associated lock-out key. The caller must call the routine once for each time it called *irq_lock()*, supplying the keys in the reverse order they were acquired, before interrupts are enabled.

This routine can only be invoked from supervisor mode. Some architectures (for example, ARM) will fail silently if invoked from user mode instead of generating an exception.

Note: This routine must also serve as a memory barrier to ensure the uniprocessor implementation of `k_spinlock_t` is correct.

Note: Can be called by ISRs.

Parameters

- `key` – Lock-out key generated by *irq_lock()*.

Returns N/A

`irq_enable(irq)`

Enable an IRQ.

This routine enables interrupts from source *irq*.

Parameters

- `irq` – IRQ line.

Returns N/A

`irq_disable(irq)`

Disable an IRQ.

This routine disables interrupts from source *irq*.

Parameters

- `irq` – IRQ line.

Returns N/A

`irq_is_enabled(irq)`

Get IRQ enable state.

This routine indicates if interrupts from source *irq* are enabled.

Parameters

- `irq` – IRQ line.

Returns interrupt enable state, true or false

Functions

static inline int `irq_connect_dynamic`(unsigned int irq, unsigned int priority, void (*routine)(const void *parameter), const void *parameter, uint32_t flags)

Configure a dynamic interrupt.

Use this instead of *IRQ_CONNECT()* if arguments cannot be known at build time.

Parameters

- `irq` – IRQ line number
- `priority` – Interrupt priority
- `routine` – Interrupt service routine
- `parameter` – ISR parameter
- `flags` – Arch-specific IRQ configuration flags

Returns The vector assigned to this interrupt

static inline unsigned int `irq_get_level`(unsigned int irq)

bool `k_is_in_isr`(void)

Determine if code is running at interrupt level.

This routine allows the caller to customize its actions, depending on whether it is a thread or an ISR.

Function properties (list may not be complete) *isr-ok*

Returns false if invoked by a thread.

Returns true if invoked by an ISR.

int `k_is_preempt_thread`(void)

Determine if code is running in a preemptible thread.

This routine allows the caller to customize its actions, depending on whether it can be preempted by another thread. The routine returns a 'true' value if all of the following conditions are met:

- The code is running in a thread, not at ISR.
- The thread's priority is in the preemptible range.
- The thread has not locked the scheduler.

Function properties (list may not be complete) *isr-ok*

Returns 0 if invoked by an ISR or by a cooperative thread.

Returns Non-zero if invoked by a preemptible thread.

static inline bool `k_is_pre_kernel`(void)

Test whether startup is in the before-main-task phase.

This routine allows the caller to customize its actions, depending on whether it being invoked before the kernel is fully active.

Function properties (list may not be complete) *isr-ok*

Returns true if invoked before post-kernel initialization

Returns false if invoked during/after post-kernel initialization

Polling API

The polling API is used to wait concurrently for any one of multiple conditions to be fulfilled.

Concepts The polling API's main function is *`k_poll()`*, which is very similar in concept to the POSIX `poll()` function, except that it operates on kernel objects rather than on file descriptors.

The polling API allows a single thread to wait concurrently for one or more conditions to be fulfilled without actively looking at each one individually.

There is a limited set of such conditions:

- a semaphore becomes available
- a kernel FIFO contains data ready to be retrieved
- a poll signal is raised

A thread that wants to wait on multiple conditions must define an array of **poll events**, one for each condition.

All events in the array must be initialized before the array can be polled on.

Each event must specify which **type** of condition must be satisfied so that its state is changed to signal the requested condition has been met.

Each event must specify what **kernel object** it wants the condition to be satisfied.

Each event must specify which **mode** of operation is used when the condition is satisfied.

Each event can optionally specify a **tag** to group multiple events together, to the user's discretion.

Apart from the kernel objects, there is also a **poll signal** pseudo-object type that be directly signaled.

The *k_poll()* function returns as soon as one of the conditions it is waiting for is fulfilled. It is possible for more than one to be fulfilled when *k_poll()* returns, if they were fulfilled before *k_poll()* was called, or due to the preemptive multi-threading nature of the kernel. The caller must look at the state of all the poll events in the array to figured out which ones were fulfilled and what actions to take.

Currently, there is only one mode of operation available: the object is not acquired. As an example, this means that when *k_poll()* returns and the poll event states that the semaphore is available, the caller of *k_poll()* must then invoke *k_sem_take()* to take ownership of the semaphore. If the semaphore is contested, there is no guarantee that it will be still available when *k_sem_give()* is called.

Implementation

Using k_poll() The main API is *k_poll()*, which operates on an array of poll events of type *k_poll_event*. Each entry in the array represents one event a call to *k_poll()* will wait for its condition to be fulfilled.

They can be initialized using either the runtime initializers *K_POLL_EVENT_INITIALIZER()* or *k_poll_event_init()*, or the static initializer *K_POLL_EVENT_STATIC_INITIALIZER()*. An object that matches the **type** specified must be passed to the initializers. The **mode** *must* be set to *K_POLL_MODE_NOTIFY_ONLY*. The state *must* be set to *K_POLL_STATE_NOT_READY* (the initializers take care of this). The user **tag** is optional and completely opaque to the API: it is there to help a user to group similar events together. Being optional, it is passed to the static initializer, but not the runtime ones for performance reasons. If using runtime initializers, the user must set it separately in the *k_poll_event* data structure. If an event in the array is to be ignored, most likely temporarily, its type can be set to K_POLL_TYPE_IGNORE.

```
struct k_poll_event events[2] = {
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_SEM_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_sem, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_fifo, 0),
};
```

or at runtime

```
struct k_poll_event events[2];
void some_init(void)
{
```

(continues on next page)

(continued from previous page)

```
    k_poll_event_init(&events[0],
                      K_POLL_TYPE_SEM_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_sem);

    k_poll_event_init(&events[1],
                      K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_fifo);

    // tags are left uninitialized if unused
}
```

After the events are initialized, the array can be passed to *k_poll()*. A timeout can be specified to wait only for a specified amount of time, or the special values *K_NO_WAIT* and *K_FOREVER* to either not wait or wait until an event condition is satisfied and not sooner.

A list of pollers is offered on each semaphore or FIFO and as many events can wait in it as the app wants. Notice that the waiters will be served in first-come-first-serve order, not in priority order.

In case of success, *k_poll()* returns 0. If it times out, it returns -*EAGAIN*.

```
// assume there is no contention on this semaphore and FIFO
// -EADDRINUSE will not occur; the semaphore and/or data will be available

void do_stuff(void)
{
    rc = k_poll(events, 2, 1000);
    if (rc == 0) {
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        } else if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        }
    } else {
        // handle timeout
    }
}
```

When *k_poll()* is called in a loop, the events state must be reset to *K_POLL_STATE_NOT_READY* by the user.

```
void do_stuff(void)
{
    for(;;) {
        rc = k_poll(events, 2, K_FOREVER);
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        } else if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        }
        events[0].state = K_POLL_STATE_NOT_READY;
        events[1].state = K_POLL_STATE_NOT_READY;
    }
}
```

Using k_poll_signal_raise() One of the types of events is *K_POLL_TYPE_SIGNAL* : this is a "direct" signal to a poll event. This can be seen as a lightweight binary semaphore only one thread can wait for.

A poll signal is a separate object of type *k_poll_signal* that must be attached to a k_poll_event, similar to a semaphore or FIFO. It must first be initialized either via *K_POLL_SIGNAL_INITIALIZER()* or *k_poll_signal_init()*.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);
}
```

It is signaled via the *k_poll_signal_raise()* function. This function takes a user **result** parameter that is opaque to the API and can be used to pass extra information to the thread waiting on the event.

```
struct k_poll_signal signal;

// thread A
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal),
    };

    k_poll(events, 1, K_FOREVER);

    if (events.signal->result == 0x1337) {
        // A-OK!
    } else {
        // weird error
    }
}

// thread B
void signal_do_stuff(void)
{
    k_poll_signal_raise(&signal, 0x1337);
}
```

If the signal is to be polled in a loop, *both* its event state and its **signaled** field *must* be reset on each iteration if it has been signaled.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal),
    };

    for (;;) {
```

(continues on next page)

(continued from previous page)

```
        k_poll(events, 1, K_FOREVER);

        if (events[0].signal->result == 0x1337) {
            // A-OK!
        } else {
            // weird error
        }

        events[0].signal->signaled = 0;
        events[0].state = K_POLL_STATE_NOT_READY;
    }
}
```

Suggested Uses Use *k_poll()* to consolidate multiple threads that would be pending on one object each, saving possibly large amounts of stack space.

Use a poll signal as a lightweight binary semaphore if only one thread pends on it.

Note: Because objects are only signaled if no other thread is waiting for them to become available and only one thread can poll on a specific object, polling is best used when objects are not subject of contention between multiple threads, basically when a single thread operates as a main "server" or "dispatcher" for multiple objects and is the only one trying to acquire these objects.

Configuration Options Related configuration options:

- CONFIG_POLL

API Reference

group poll_apis

Defines

K_POLL_TYPE_IGNORE

K_POLL_TYPE_SIGNAL

K_POLL_TYPE_SEM_AVAILABLE

K_POLL_TYPE_DATA_AVAILABLE

K_POLL_TYPE_FIFO_DATA_AVAILABLE

K_POLL_TYPE_MSGQ_DATA_AVAILABLE

K_POLL_STATE_NOT_READY

K_POLL_STATE_SIGNALED

K_POLL_STATE_SEM_AVAILABLE

K_POLL_STATE_DATA_AVAILABLE

K_POLL_STATE_FIFO_DATA_AVAILABLE

K_POLL_STATE_MSGQ_DATA_AVAILABLE

K_POLL_STATE_CANCELLED

K_POLL_SIGNAL_INITIALIZER(obj)

K_POLL_EVENT_INITIALIZER(_event_type, _event_mode, _event_obj)

K_POLL_EVENT_STATIC_INITIALIZER(_event_type, _event_mode, _event_obj, event_tag)

Enums

enum k_poll_modes

Values:

enumerator K_POLL_MODE_NOTIFY_ONLY = 0

enumerator K_POLL_NUM_MODES

Functions

void k_poll_event_init(struct *k_poll_event* *event, uint32_t type, int mode, void *obj)

Initialize one struct *k_poll_event* instance.

After this routine is called on a poll event, the event it ready to be placed in an event array to be passed to *k_poll()*.

Parameters

- event – The event to initialize.
- type – A bitfield of the types of event, from the K_POLL_TYPE_xxx values. Only values that apply to the same object being polled can be used together. Choosing K_POLL_TYPE_IGNORE disables the event.
- mode – Future. Use K_POLL_MODE_NOTIFY_ONLY.
- obj – Kernel object or poll signal.

Returns N/A

int k_poll(struct *k_poll_event* *events, int num_events, *k_timeout_t* timeout)

Wait for one or many of multiple poll events to occur.

This routine allows a thread to wait concurrently for one or many of multiple poll events to have occurred. Such events can be a kernel object being available, like a semaphore, or a poll signal event.

When an event notifies that a kernel object is available, the kernel object is not "given" to the thread calling *k_poll()*: it merely signals the fact that the object was available when the *k_poll()* call was in effect. Also, all threads trying to acquire an object the regular way, i.e. by pending on the object, have precedence over the thread polling on the object. This means that the polling thread will never get the poll event on an object until the object becomes available and its pend queue is empty. For this reason, the *k_poll()* call is more effective when the objects being polled only have one thread, the polling thread, trying to acquire them.

When *k_poll()* returns 0, the caller should loop on all the events that were passed to *k_poll()* and check the state field for the values that were expected and take the associated actions.

Before being reused for another call to *k_poll()*, the user has to reset the state field to K_POLL_STATE_NOT_READY.

When called from user mode, a temporary memory allocation is required from the caller's resource pool.

Parameters

- `events` – An array of events to be polled for.
- `num_events` – The number of events in the array.
- `timeout` – Waiting period for an event to be ready, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – One or more events are ready.
- `-EAGAIN` – Waiting period timed out.
- `-EINTR` – Polling has been interrupted, e.g. with *k_queue_cancel_wait()*. All output events are still set and valid, cancelled event(s) will be set to K_POLL_STATE_CANCELLED. In other words, -EINTR status means that at least one of output events is K_POLL_STATE_CANCELLED.
- `-ENOMEM` – Thread resource pool insufficient memory (user mode only)
- `-EINVAL` – Bad parameters (user mode only)

void `k_poll_signal_init`(struct *k_poll_signal* *sig)

Initialize a poll signal object.

Ready a poll signal object to be signaled via *k_poll_signal_raise()*.

Parameters

- `sig` – A poll signal.

Returns N/A

void `k_poll_signal_reset`(struct *k_poll_signal* *sig)

void `k_poll_signal_check`(struct *k_poll_signal* *sig, unsigned int *signaled, int *result)

Fetch the signaled state and result value of a poll signal.

Parameters

- `sig` – A poll signal object
- `signaled` – An integer buffer which will be written nonzero if the object was signaled
- `result` – An integer destination buffer which will be written with the result value if the object was signaled, or an undefined value if it was not.

int k_poll_signal_raise(struct *k_poll_signal* *sig, int result)

Signal a poll signal object.

This routine makes ready a poll signal, which is basically a poll event of type K_POLL_TYPE_SIGNAL. If a thread was polling on that event, it will be made ready to run. A *result* value can be specified.

The poll signal contains a 'signaled' field that, when set by *k_poll_signal_raise()*, stays set until the user sets it back to 0 with *k_poll_signal_reset()*. It thus has to be reset by the user before being passed again to *k_poll()* or *k_poll()* will consider it being signaled, and will return immediately.

Note: The result is stored and the 'signaled' field is set even if this function returns an error indicating that an expiring poll was not notified. The next *k_poll()* will detect the missed raise.

Parameters

- sig – A poll signal.
- result – The value to store in the result field of the signal.

Return values

- 0 – The signal was delivered successfully.
- -EAGAIN – The polling thread's timeout is in the process of expiring.

struct k_poll_signal

#include <kernel.h>

Public Members

sys_dlist_t poll_events

PRIVATE - DO NOT TOUCH

unsigned int signaled

1 if the event has been signaled, 0 otherwise. Stays set to 1 until user resets it to 0.

int result

custom result value passed to *k_poll_signal_raise()* if needed

struct k_poll_event

#include <kernel.h> Poll Event.

Public Members

struct z_poller *poller

PRIVATE - DO NOT TOUCH

uint32_t tag

optional user-specified tag, opaque, untouched by the API

uint32_t `type`
: bitfield of event types (bitwise-ORed K_POLL_TYPE_xxx values)

uint32_t `state`
: bitfield of event states (bitwise-ORed K_POLL_STATE_xxx values)

uint32_t `mode`
: mode of operation, from enum k_poll_modes

uint32_t `unused`
: unused bits in 32-bit word

union *k_poll_event*.[anonymous] `[anonymous]`
: per-type data

Semaphores

A *semaphore* is a kernel object that implements a traditional counting semaphore.

Concepts Any number of semaphores can be defined (limited only by available RAM). Each semaphore is referenced by its memory address.

A semaphore has the following key properties:

- A **count** that indicates the number of times the semaphore can be taken. A count of zero indicates that the semaphore is unavailable.
- A **limit** that indicates the maximum value the semaphore's count can reach.

A semaphore must be initialized before it can be used. Its count must be set to a non-negative value that is less than or equal to its limit.

A semaphore may be **given** by a thread or an ISR. Giving the semaphore increments its count, unless the count is already equal to the limit.

A semaphore may be **taken** by a thread. Taking the semaphore decrements its count, unless the semaphore is unavailable (i.e. at zero). When a semaphore is unavailable a thread may choose to wait for it to be given. Any number of threads may wait on an unavailable semaphore simultaneously. When the semaphore is given, it is taken by the highest priority thread that has waited longest.

Note: You may initialize a "full" semaphore (count equal to limit) to limit the number of threads able to execute the critical section at the same time. You may also initialize an empty semaphore (count equal to 0, with a limit greater than 0) to create a gate through which no waiting thread may pass until the semaphore is incremented. All standard use cases of the common semaphore are supported.

Note: The kernel does allow an ISR to take a semaphore, however the ISR must not attempt to wait if the semaphore is unavailable.

Implementation

Defining a Semaphore A semaphore is defined using a variable of type `k_sem`. It must then be initialized by calling *k_sem_init()*.

The following code defines a semaphore, then configures it as a binary semaphore by setting its count to 0 and its limit to 1.

```
struct k_sem my_sem;

k_sem_init(&my_sem, 0, 1);
```

Alternatively, a semaphore can be defined and initialized at compile time by calling *K_SEM_DEFINE*.

The following code has the same effect as the code segment above.

```
K_SEM_DEFINE(my_sem, 0, 1);
```

Giving a Semaphore A semaphore is given by calling *k_sem_give()*.

The following code builds on the example above, and gives the semaphore to indicate that a unit of data is available for processing by a consumer thread.

```
void input_data_interrupt_handler(void *arg)
{
    /* notify thread that data is available */
    k_sem_give(&my_sem);

    ...
}
```

Taking a Semaphore A semaphore is taken by calling *k_sem_take()*.

The following code builds on the example above, and waits up to 50 milliseconds for the semaphore to be given. A warning is issued if the semaphore is not obtained in time.

```
void consumer_thread(void)
{
    ...

    if (k_sem_take(&my_sem, K_MSEC(50)) != 0) {
        printk("Input data not available!");
    } else {
        /* fetch available data */
        ...
```

(continues on next page)

(continued from previous page)

```
    }
    ...
}
```

Suggested Uses Use a semaphore to control access to a set of resources by multiple threads.

Use a semaphore to synchronize processing between a producing and consuming threads or ISRs.

Configuration Options Related configuration options:

- None.

API Reference

group `semaphore_apis`

Defines

`K_SEM_MAX_LIMIT`

Maximum limit value allowed for a semaphore.

This is intended for use when a semaphore does not have an explicit maximum limit, and instead is just used for counting purposes.

`K_SEM_DEFINE(name, initial_count, count_limit)`

Statically define and initialize a semaphore.

The semaphore can be accessed outside the module where it is defined using:

```
extern struct k_sem <name>;
```

Parameters

- `name` – Name of the semaphore.
- `initial_count` – Initial semaphore count.
- `count_limit` – Maximum permitted semaphore count.

Functions

int `k_sem_init(struct k_sem *sem, unsigned int initial_count, unsigned int limit)`

Initialize a semaphore.

This routine initializes a semaphore object, prior to its first use.

See also:

K_SEM_MAX_LIMIT

Parameters

- `sem` – Address of the semaphore.
- `initial_count` – Initial semaphore count.

- `limit` – Maximum permitted semaphore count.

Return values

- `0` – Semaphore created successfully
- `-EINVAL` – Invalid values

int `k_sem_take`(struct k_sem *sem, *k_timeout_t* timeout)

Take a semaphore.

This routine takes *sem*.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `sem` – Address of the semaphore.
- `timeout` – Waiting period to take the semaphore, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Semaphore taken.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out, or the semaphore was reset during the waiting period.

void `k_sem_give`(struct k_sem *sem)

Give a semaphore.

This routine gives *sem*, unless the semaphore is already at its maximum permitted count.

Function properties (list may not be complete) *isr-ok*

Parameters

- `sem` – Address of the semaphore.

Returns N/A

void `k_sem_reset`(struct k_sem *sem)

Resets a semaphore's count to zero.

This routine sets the count of *sem* to zero. Any outstanding semaphore takes will be aborted with -EAGAIN.

Parameters

- `sem` – Address of the semaphore.

Returns N/A

unsigned int `k_sem_count_get`(struct k_sem *sem)

Get a semaphore's count.

This routine returns the current count of *sem*.

Parameters

- `sem` – Address of the semaphore.

Returns Current semaphore count.

User Mode Semaphore API Reference The sys_sem exists in user memory working as counter semaphore for user mode thread when user mode enabled. When user mode isn't enabled, sys_sem behaves like k_sem.

group `user_semaphore_apis`

Defines

`SYS_SEM_DEFINE(_name, _initial_count, _count_limit)`

Statically define and initialize a sys_sem.

The semaphore can be accessed outside the module where it is defined using:

```
extern struct sys_sem <name>;
```

Route this to memory domains using K_APP_DMEM().

Parameters

- `_name` – Name of the semaphore.
- `_initial_count` – Initial semaphore count.
- `_count_limit` – Maximum permitted semaphore count.

Functions

int `sys_sem_init(struct sys_sem *sem, unsigned int initial_count, unsigned int limit)`

Initialize a semaphore.

This routine initializes a semaphore instance, prior to its first use.

Parameters

- `sem` – Address of the semaphore.
- `initial_count` – Initial semaphore count.
- `limit` – Maximum permitted semaphore count.

Return values

- `0` – Initial success.
- `-EINVAL` – Bad parameters, the value of limit should be located in (0, INT_MAX] and initial_count shouldn't be greater than limit.

int `sys_sem_give(struct sys_sem *sem)`

Give a semaphore.

This routine gives *sem*, unless the semaphore is already at its maximum permitted count.

Parameters

- `sem` – Address of the semaphore.

Return values

- `0` – Semaphore given.
- `-EINVAL` – Parameter address not recognized.

- -EACCES – Caller does not have enough access.
- -EAGAIN – Count reached Maximum permitted count and try again.

int sys_sem_take(struct sys_sem *sem, *k_timeout_t* timeout)

Take a sys_sem.

This routine takes *sem*.

Parameters

- sem – Address of the sys_sem.
- timeout – Waiting period to take the sys_sem, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- 0 – sys_sem taken.
- -EINVAL – Parameter address not recognized.
- -ETIMEDOUT – Waiting period timed out.
- -EACCES – Caller does not have enough access.

unsigned int sys_sem_count_get(struct sys_sem *sem)

Get sys_sem's value.

This routine returns the current value of *sem*.

Parameters

- sem – Address of the sys_sem.

Returns Current value of sys_sem.

Mutexes

A *mutex* is a kernel object that implements a traditional reentrant mutex. A mutex allows multiple threads to safely share an associated hardware or software resource by ensuring mutually exclusive access to the resource.

Concepts Any number of mutexes can be defined (limited only by available RAM). Each mutex is referenced by its memory address.

A mutex has the following key properties:

- A **lock count** that indicates the number of times the mutex has be locked by the thread that has locked it. A count of zero indicates that the mutex is unlocked.
- An **owning thread** that identifies the thread that has locked the mutex, when it is locked.

A mutex must be initialized before it can be used. This sets its lock count to zero.

A thread that needs to use a shared resource must first gain exclusive rights to access it by **locking** the associated mutex. If the mutex is already locked by another thread, the requesting thread may choose to wait for the mutex to be unlocked.

After locking a mutex, the thread may safely use the associated resource for as long as needed; however, it is considered good practice to hold the lock for as short a time as possible to avoid negatively impacting other threads that want to use the resource. When the thread no longer needs the resource it must **unlock** the mutex to allow other threads to use the resource.

Any number of threads may wait on a locked mutex simultaneously. When the mutex becomes unlocked it is then locked by the highest-priority thread that has waited the longest.

Note: Mutex objects are *not* designed for use by ISRs.

Reentrant Locking A thread is permitted to lock a mutex it has already locked. This allows the thread to access the associated resource at a point in its execution when the mutex may or may not already be locked.

A mutex that is repeatedly locked by a thread must be unlocked an equal number of times before the mutex becomes fully unlocked so it can be claimed by another thread.

Priority Inheritance The thread that has locked a mutex is eligible for *priority inheritance*. This means the kernel will *temporarily* elevate the thread's priority if a higher priority thread begins waiting on the mutex. This allows the owning thread to complete its work and release the mutex more rapidly by executing at the same priority as the waiting thread. Once the mutex has been unlocked, the unlocking thread resets its priority to the level it had before locking that mutex.

Note: The `CONFIG_PRIORITY_CEILING` configuration option limits how high the kernel can raise a thread's priority due to priority inheritance. The default value of 0 permits unlimited elevation.

When two or more threads wait on a mutex held by a lower priority thread, the kernel adjusts the owning thread's priority each time a thread begins waiting (or gives up waiting). When the mutex is eventually unlocked, the unlocking thread's priority correctly reverts to its original non-elevated priority.

The kernel does *not* fully support priority inheritance when a thread holds two or more mutexes simultaneously. This situation can result in the thread's priority not reverting to its original non-elevated priority when all mutexes have been released. It is recommended that a thread lock only a single mutex at a time when multiple mutexes are shared between threads of different priorities.

Implementation

Defining a Mutex A mutex is defined using a variable of type *k_mutex*. It must then be initialized by calling *k_mutex_init()*.

The following code defines and initializes a mutex.

```
struct k_mutex my_mutex;

k_mutex_init(&my_mutex);
```

Alternatively, a mutex can be defined and initialized at compile time by calling *K_MUTEX_DEFINE*.

The following code has the same effect as the code segment above.

```
K_MUTEX_DEFINE(my_mutex);
```

Locking a Mutex A mutex is locked by calling *k_mutex_lock()*.

The following code builds on the example above, and waits indefinitely for the mutex to become available if it is already locked by another thread.

```
k_mutex_lock(&my_mutex, K_FOREVER);
```

The following code waits up to 100 milliseconds for the mutex to become available, and gives a warning if the mutex does not become available.

```
if (k_mutex_lock(&my_mutex, K_MSEC(100)) == 0) {
    /* mutex successfully locked */
} else {
    printf("Cannot lock XYZ display\n");
}
```

Unlocking a Mutex A mutex is unlocked by calling *k_mutex_unlock()*.

The following code builds on the example above, and unlocks the mutex that was previously locked by the thread.

```
k_mutex_unlock(&my_mutex);
```

Suggested Uses Use a mutex to provide exclusive access to a resource, such as a physical device.

Configuration Options Related configuration options:

- CONFIG_PRIORITY_CEILING

API Reference

group mutex_apis

Defines

K_MUTEX_DEFINE(name)

Statically define and initialize a mutex.

The mutex can be accessed outside the module where it is defined using:

```
extern struct k_mutex <name>;
```

Parameters

- name – Name of the mutex.

Functions

int `k_mutex_init`(struct *k_mutex* *mutex)

Initialize a mutex.

This routine initializes a mutex object, prior to its first use.

Upon completion, the mutex is available and does not have an owner.

Parameters

- `mutex` – Address of the mutex.

Return values `0` – Mutex object created

int `k_mutex_lock`(struct *k_mutex* *mutex, *k_timeout_t* timeout)

Lock a mutex.

This routine locks *mutex*. If the mutex is locked by another thread, the calling thread waits until the mutex becomes available or until a timeout occurs.

A thread is permitted to lock a mutex it has already locked. The operation completes immediately and the lock count is increased by 1.

Mutexes may not be locked in ISRs.

Parameters

- `mutex` – Address of the mutex.
- `timeout` – Waiting period to lock the mutex, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Mutex locked.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

int `k_mutex_unlock`(struct *k_mutex* *mutex)

Unlock a mutex.

This routine unlocks *mutex*. The mutex must already be locked by the calling thread.

The mutex cannot be claimed by another thread until it has been unlocked by the calling thread as many times as it was previously locked by that thread.

Mutexes may not be unlocked in ISRs, as mutexes must only be manipulated in thread context due to ownership and priority inheritance semantics.

Parameters

- `mutex` – Address of the mutex.

Return values

- `0` – Mutex unlocked.
- `-EPERM` – The current thread does not own the mutex
- `-EINVAL` – The mutex is not locked

struct `k_mutex`

#include <kernel.h> Mutex Structure

Public Members

_wait_q_t `wait_q`

Mutex wait queue

struct *k_thread* *`owner`

Mutex owner

uint32_t `lock_count`

Current lock count

int `owner_orig_prio`

Original thread priority

Futex API Reference k_futex is a lightweight mutual exclusion primitive designed to minimize kernel involvement. Uncontended operation relies only on atomic access to shared memory. k_futex are tracked as kernel objects and can live in user memory so that any access bypasses the kernel object permission management mechanism.

group `futex_apis`

Functions

int `k_futex_wait`(struct k_futex *futex, int expected, *k_timeout_t* timeout)

Pend the current thread on a futex.

Tests that the supplied futex contains the expected value, and if so, goes to sleep until some other thread calls *k_futex_wake()* on it.

Parameters

- `futex` – Address of the futex.
- `expected` – Expected value of the futex, if it is different the caller will not wait on it.
- `timeout` – Non-negative waiting period on the futex, or one of the special values K_NO_WAIT or K_FOREVER.

Return values

- `-EACCES` – Caller does not have read access to futex address.
- `-EAGAIN` – If the futex value did not match the expected parameter.
- `-EINVAL` – Futex parameter address not recognized by the kernel.
- `-ETIMEDOUT` – Thread woke up due to timeout and not a futex wakeup.
- `0` – if the caller went to sleep and was woken up. The caller should check the futex's value on wakeup to determine if it needs to block again.

int `k_futex_wake`(struct k_futex *futex, bool wake_all)

Wake one/all threads pending on a futex.

Wake up the highest priority thread pending on the supplied futex, or wakeup all the threads pending on the supplied futex, and the behavior depends on wake_all.

Parameters

- `futex` – Futex to wake up pending threads.
- `wake_all` – If true, wake up all pending threads; If false, wakeup the highest priority thread.

Return values

- `-EACCES` – Caller does not have access to the futex address.
- `-EINVAL` – Futex parameter address not recognized by the kernel.
- `Number` – of threads that were woken up.

User Mode Mutex API Reference sys_mutex behaves almost exactly like k_mutex, with the added advantage that a sys_mutex instance can reside in user memory. When user mode isn't enabled, sys_mutex behaves like k_mutex.

group `user_mutex_apis`

Defines

`SYS_MUTEX_DEFINE(name)`

Statically define and initialize a sys_mutex.

The mutex can be accessed outside the module where it is defined using:

```
extern struct sys_mutex <name>;
```

Route this to memory domains using K_APP_DMEM().

Parameters

- `name` – Name of the mutex.

Functions

static inline void `sys_mutex_init(struct sys_mutex *mutex)`

Initialize a mutex.

This routine initializes a mutex object, prior to its first use.

Upon completion, the mutex is available and does not have an owner.

This routine is only necessary to call when userspace is disabled and the mutex was not created with *SYS_MUTEX_DEFINE()*.

Parameters

- `mutex` – Address of the mutex.

Returns N/A

static inline int `sys_mutex_lock(struct sys_mutex *mutex, k_timeout_t timeout)`

Lock a mutex.

This routine locks *mutex*. If the mutex is locked by another thread, the calling thread waits until the mutex becomes available or until a timeout occurs.

A thread is permitted to lock a mutex it has already locked. The operation completes immediately and the lock count is increased by 1.

Parameters

- `mutex` – Address of the mutex, which may reside in user memory

- `timeout` – Waiting period to lock the mutex, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Mutex locked.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.
- `-EACCES` – Caller has no access to provided mutex address
- `-EINVAL` – Provided mutex not recognized by the kernel

static inline int `sys_mutex_unlock`(struct sys_mutex *mutex)

Unlock a mutex.

This routine unlocks *mutex*. The mutex must already be locked by the calling thread.

The mutex cannot be claimed by another thread until it has been unlocked by the calling thread as many times as it was previously locked by that thread.

Parameters

- `mutex` – Address of the mutex, which may reside in user memory

Return values

- `0` – Mutex unlocked
- `-EACCES` – Caller has no access to provided mutex address
- `-EINVAL` – Provided mutex not recognized by the kernel or mutex wasn't locked
- `-EPERM` – Caller does not own the mutex

Condition Variables

A *condition variable* is a synchronization primitive that enables threads to wait until a particular condition occurs.

Concepts Any number of condition variables can be defined (limited only by available RAM). Each condition variable is referenced by its memory address.

To wait for a condition to become true, a thread can make use of a condition variable.

A condition variable is basically a queue of threads that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition). The function *`k_condvar_wait()`* performs atomically the following steps;

1. Releases the last acquired mutex.

2. Puts the current thread in the condition variable queue.

Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue by signaling on the condition using *k_condvar_signal()* or *k_condvar_broadcast()* then it:

1. Re-acquires the mutex previously released.
2. Returns from *k_condvar_wait()*.

A condition variable must be initialized before it can be used.

Implementation

Defining a Condition Variable A condition variable is defined using a variable of type `k_condvar`. It must then be initialized by calling *k_condvar_init()*.

The following code defines a condition variable:

```
struct k_condvar my_condvar;

k_condvar_init(&my_condvar);
```

Alternatively, a condition variable can be defined and initialized at compile time by calling *K_CONDVAR_DEFINE*.

The following code has the same effect as the code segment above.

```
K_CONDVAR_DEFINE(my_condvar);
```

Waiting on a Condition Variable A thread can wait on a condition by calling *k_condvar_wait()*.

The following code waits on the condition variable.

```
K_MUTEX_DEFINE(mutex);
K_CONDVAR_DEFINE(condvar)

void main(void)
{
    k_mutex_lock(&mutex, K_FOREVER);

    /* block this thread until another thread signals cond. While
     * blocked, the mutex is released, then re-acquired before this
     * thread is woken up and the call returns.
     */
    k_condvar_wait(&condvar, &mutex, K_FOREVER);
    ...
    k_mutex_unlock(&mutex);
}
```

Signaling a Condition Variable A condition variable is signaled on by calling *k_condvar_signal()* for one thread or by calling *k_condvar_broadcast()* for multiple threads.

The following code builds on the example above.

```
void worker_thread(void)
{
    k_mutex_lock(&mutex, K_FOREVER);
```

(continues on next page)

(continued from previous page)

```
    /*
     * Do some work and fullfill the condition
     */
    ...
    ...
    k_condvar_signal(&condvar);
    k_mutex_unlock(&mutex);
}
```

Suggested Uses Use condition variables with a mutex to signal changing states (conditions) from one thread to another thread. Condition variables are not the condition itself and they are not events. The condition is contained in the surrounding programming logic.

Mutexes alone are not designed for use as a notification/synchronization mechanism. They are meant to provide mutually exclusive access to a shared resource only.

Configuration Options Related configuration options:

- None.

API Reference

group `condvar_apis`

Defines

`K_CONDVAR_DEFINE(name)`

Statically define and initialize a condition variable.

The condition variable can be accessed outside the module where it is defined using:

```
extern struct k_condvar <name>;
```

Parameters

- `name` – Name of the condition variable.

Functions

int `k_condvar_init(struct k_condvar *condvar)`

Initialize a condition variable.

Parameters

- `condvar` – pointer to a `k_condvar` structure

Return values `0` – Condition variable created successfully

int `k_condvar_signal(struct k_condvar *condvar)`

Signals one thread that is pending on the condition variable.

Parameters

- `condvar` – pointer to a `k_condvar` structure

Return values `0` – On success

int `k_condvar_broadcast`(struct k_condvar *condvar)

Unblock all threads that are pending on the condition variable.

Parameters

- `condvar` – pointer to a `k_condvar` structure

Returns An integer with number of woken threads on success

int `k_condvar_wait`(struct k_condvar *condvar, struct *k_mutex* *mutex, *k_timeout_t* timeout)

Waits on the condition variable releasing the mutex lock.

Automically releases the currently owned mutex, blocks the current thread waiting on the condition variable specified by *condvar*, and finally acquires the mutex again.

The waiting thread unblocks only after another thread calls k_condvar_signal, or k_condvar_broadcast with the same condition variable.

Parameters

- `condvar` – pointer to a `k_condvar` structure
- `mutex` – Address of the mutex.
- `timeout` – Waiting period for the condition variable or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – On success
- `-EAGAIN` – Waiting period timed out.

Symmetric Multiprocessing

On multiprocessor architectures, Zephyr supports the use of multiple physical CPUs running Zephyr application code. This support is "symmetric" in the sense that no specific CPU is treated specially by default. Any processor is capable of running any Zephyr thread, with access to all standard Zephyr APIs supported.

No special application code needs to be written to take advantage of this feature. If there are two Zephyr application threads runnable on a supported dual processor device, they will both run simultaneously.

SMP configuration is controlled under the `CONFIG_SMP` kconfig variable. This must be set to "y" to enable SMP features, otherwise a uniprocessor kernel will be built. In general the platform default will have enabled this anywhere it's supported. When enabled, the number of physical CPUs available is visible at build time as `CONFIG_MP_NUM_CPUS`. Likewise, the default for this will be the number of available CPUs on the platform and it is not expected that typical apps will change it. But it is legal and supported to set this to a smaller (but obviously not larger) number for special purposes (e.g. for testing, or to reserve a physical CPU for running non-Zephyr code).

Synchronization At the application level, core Zephyr IPC and synchronization primitives all behave identically under an SMP kernel. For example semaphores used to implement blocking mutual exclusion continue to be a proper application choice.

At the lowest level, however, Zephyr code has often used the *irq_lock()*/*irq_unlock()* primitives to implement fine grained critical sections using interrupt masking. These APIs continue to work via an emulation layer (see below), but the masking technique does not: the fact that your CPU will not be interrupted while you are in your critical section says nothing about whether a different CPU will be running simultaneously and be inspecting or modifying the same data!

Spinlocks SMP systems provide a more constrained *k_spin_lock()* primitive that not only masks interrupts locally, as done by *irq_lock()*, but also atomically validates that a shared lock variable has been modified before returning to the caller, "spinning" on the check if needed to wait for the other CPU to exit the lock. The default Zephyr implementation of *k_spin_lock()* and *k_spin_unlock()* is built on top of the pre-existing `atomic_` layer (itself usually implemented using compiler intrinsics), though facilities exist for architectures to define their own for performance reasons.

One important difference between IRQ locks and spinlocks is that the earlier API was naturally recursive: the lock was global, so it was legal to acquire a nested lock inside of a critical section. Spinlocks are separable: you can have many locks for separate subsystems or data structures, preventing CPUs from contending on a single global resource. But that means that spinlocks must not be used recursively. Code that holds a specific lock must not try to re-acquire it or it will deadlock (it is perfectly legal to nest **distinct** spinlocks, however). A validation layer is available to detect and report bugs like this.

When used on a uniprocessor system, the data component of the spinlock (the atomic lock variable) is unnecessary and elided. Except for the recursive semantics above, spinlocks in single-CPU contexts produce identical code to legacy IRQ locks. In fact the entirety of the Zephyr core kernel has now been ported to use spinlocks exclusively.

Legacy irq_lock() emulation For the benefit of applications written to the uniprocessor locking API, *irq_lock()* and *irq_unlock()* continue to work compatibly on SMP systems with identical semantics to their legacy versions. They are implemented as a single global spinlock, with a nesting count and the ability to be atomically reacquired on context switch into locked threads. The kernel will ensure that only one thread across all CPUs can hold the lock at any time, that it is released on context switch, and that it is re-acquired when necessary to restore the lock state when a thread is switched in. Other CPUs will spin waiting for the release to happen.

The overhead involved in this process has measurable performance impact, however. Unlike uniprocessor apps, SMP apps using *irq_lock()* are not simply invoking a very short (often ~1 instruction) interrupt masking operation. That, and the fact that the IRQ lock is global, means that code expecting to be run in an SMP context should be using the spinlock API wherever possible.

CPU Mask It is often desirable for real time applications to deliberately partition work across physical CPUs instead of relying solely on the kernel scheduler to decide on which threads to execute. Zephyr provides an API, controlled by the `CONFIG_SCHED_CPU_MASK` kconfig variable, which can associate a specific set of CPUs with each thread, indicating on which CPUs it can run.

By default, new threads can run on any CPU. Calling *k_thread_cpu_mask_disable()* with a particular CPU ID will prevent that thread from running on that CPU in the future. Likewise *k_thread_cpu_mask_enable()* will re-enable execution. There are also *k_thread_cpu_mask_clear()* and *k_thread_cpu_mask_enable_all()* APIs available for convenience. For obvious reasons, these APIs are illegal if called on a runnable thread. The thread must be blocked or suspended, otherwise an `-EINVAL` will be returned.

Note that when this feature is enabled, the scheduler algorithm involved in doing the per-CPU mask test requires that the list be traversed in full. The kernel does not keep a per-CPU run queue. That means that the performance benefits from the `CONFIG_SCHED_SCALABLE` and `CONFIG_SCHED_MULTIQ` scheduler backends cannot be realized. CPU mask processing is available only when `CONFIG_SCHED_DUMB` is the selected backend. This requirement is enforced in the configuration layer.

SMP Boot Process A Zephyr SMP kernel begins boot identically to a uniprocessor kernel. Auxiliary CPUs begin in a disabled state in the architecture layer. All standard kernel initialization, including device initialization, happens on a single CPU before other CPUs are brought online.

Just before entering the application `main()` function, the kernel calls `z_smp_init()` to launch the SMP initialization process. This enumerates over the configured CPUs, calling into the architecture layer using *arch_start_cpu()* for each one. This function is passed a memory region to use as a stack on the foreign CPU (in practice it uses the area that will become that CPU's interrupt stack), the address of a

local `smp_init_top()` callback function to run on that CPU, and a pointer to a "start flag" address which will be used as an atomic signal.

The local SMP initialization (`smp_init_top()`) on each CPU is then invoked by the architecture layer. Note that interrupts are still masked at this point. This routine is responsible for calling `smp_timer_init()` to set up any needed stat in the timer driver. On many architectures the timer is a per-CPU device and needs to be configured specially on auxiliary CPUs. Then it waits (spinning) for the atomic "start flag" to be released in the main thread, to guarantee that all SMP initialization is complete before any Zephyr application code runs, and finally calls `z_swap()` to transfer control to the appropriate runnable thread via the standard scheduler API.

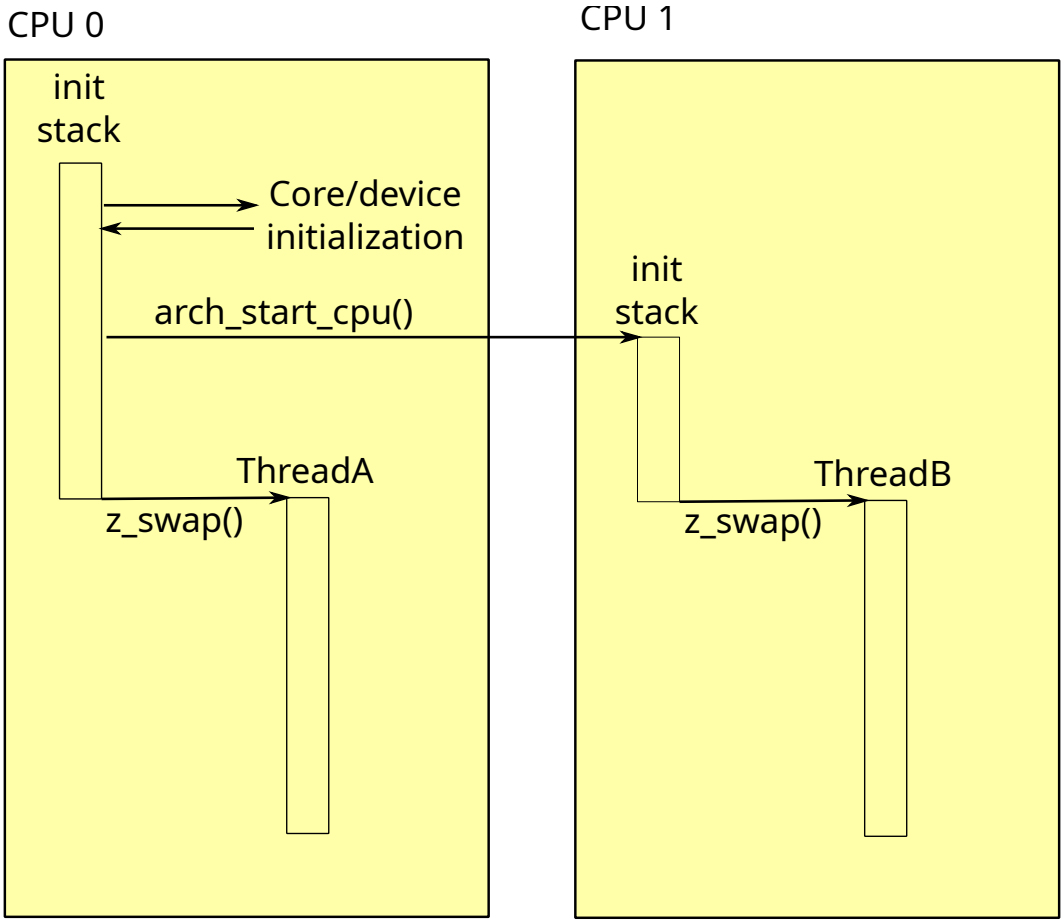


Fig. 2: Example SMP initialization process, showing a configuration with two CPUs and two app threads which begin operating simultaneously.

Interprocessor Interrupts When running in multiprocessor environments, it is occasionally the case that state modified on the local CPU needs to be synchronously handled on a different processor.

One example is the Zephyr *`k_thread_abort()`* API, which cannot return until the thread that had been aborted is no longer runnable. If it is currently running on another CPU, that becomes difficult to implement.

Another is low power idle. It is a firm requirement on many devices that system idle be implemented using a low-power mode with as many interrupts (including periodic timer interrupts) disabled or deferred as is possible. If a CPU is in such a state, and on another CPU a thread becomes runnable, the idle CPU has no way to "wake up" to handle the newly-runnable load.

So where possible, Zephyr SMP architectures should implement an interprocessor interrupt. The current framework is very simple: the architecture provides a *`arch_sched_ipi()`* call, which when invoked will flag an interrupt on all CPUs (except the current one, though that is allowed behavior) which will then

invoke the `z_sched_ipi()` function implemented in the scheduler. The expectation is that these APIs will evolve over time to encompass more functionality (e.g. cross-CPU calls), and that the scheduler-specific calls here will be implemented in terms of a more general framework.

Note that not all SMP architectures will have a usable IPI mechanism (either missing, or just undocumented/unimplemented). In those cases Zephyr provides fallback behavior that is correct, but perhaps suboptimal.

Using this, *k_thread_abort()* becomes only slightly more complicated in SMP: for the case where a thread is actually running on another CPU (we can detect this atomically inside the scheduler), we broadcast an IPI and spin, waiting for the thread to either become "DEAD" or for it to re-enter the queue (in which case we terminate it the same way we would have in uniprocessor mode). Note that the "aborted" check happens on any interrupt exit, so there is no special handling needed in the IPI per se. This allows us to implement a reasonable fallback when IPI is not available: we can simply spin, waiting until the foreign CPU receives any interrupt, though this may be a much longer time!

Likewise idle wakeups are trivially implementable with an empty IPI handler. If a thread is added to an empty run queue (i.e. there may have been idle CPUs), we broadcast an IPI. A foreign CPU will then be able to see the new thread when exiting from the interrupt and will switch to it if available.

Without an IPI, however, a low power idle that requires an interrupt will not work to synchronously run new threads. The workaround in that case is more invasive: Zephyr will **not** enter the system idle handler and will instead spin in its idle loop, testing the scheduler state at high frequency (not spinning on it though, as that would involve severe lock contention) for new threads. The expectation is that power constrained SMP applications are always going to provide an IPI, and this code will only be used for testing purposes or on systems without power consumption requirements.

SMP Kernel Internals In general, Zephyr kernel code is SMP-agnostic and, like application code, will work correctly regardless of the number of CPUs available. But in a few areas there are notable changes in structure or behavior.

Per-CPU data Many elements of the core kernel data need to be implemented for each CPU in SMP mode. For example, the `_current` thread pointer obviously needs to reflect what is running locally, there are many threads running concurrently. Likewise a kernel-provided interrupt stack needs to be created and assigned for each physical CPU, as does the interrupt nesting count used to detect ISR state.

These fields are now moved into a separate struct `_cpu` instance within the `_kernel` struct, which has a `cpus[]` array indexed by ID. Compatibility fields are provided for legacy uniprocessor code trying to access the fields of `cpus[0]` using the older syntax and assembly offsets.

Note that an important requirement on the architecture layer is that the pointer to this CPU struct be available rapidly when in kernel context. The expectation is that *arch_curr_cpu()* will be implemented using a CPU-provided register or addressing mode that can store this value across arbitrary context switches or interrupts and make it available to any kernel-mode code.

Similarly, where on a uniprocessor system Zephyr could simply create a global "idle thread" at the lowest priority, in SMP we may need one for each CPU. This makes the internal predicate test for "_is_idle()" in the scheduler, which is a hot path performance environment, more complicated than simply testing the thread pointer for equality with a known static variable. In SMP mode, idle threads are distinguished by a separate field in the thread struct.

Switch-based context switching The traditional Zephyr context switch primitive has been `z_swap()`. Unfortunately, this function takes no argument specifying a thread to switch to. The expectation has always been that the scheduler has already made its preemption decision when its state was last modified and cached the resulting "next thread" pointer in a location where architecture context switch primitives can find it via a simple struct offset. That technique will not work in SMP, because the other CPU may have modified scheduler state since the current CPU last exited the scheduler (for example: it might already be running that cached thread!).

Instead, the SMP "switch to" decision needs to be made synchronously with the swap call, and as we don't want per-architecture assembly code to be handling scheduler internal state, Zephyr requires a somewhat lower-level context switch primitives for SMP systems: *arch_switch()* is always called with interrupts masked, and takes exactly two arguments. The first is an opaque (architecture defined) handle to the context to which it should switch, and the second is a pointer to such a handle into which it should store the handle resulting from the thread that is being switched out.

The kernel then implements a portable `z_swap()` implementation on top of this primitive which includes the relevant scheduler logic in a location where the architecture doesn't need to understand it. Similarly, on interrupt exit, switch-based architectures are expected to call `z_get_next_switch_handle()` to retrieve the next thread to run from the scheduler, passing in an "interrupted" handle reflecting the same opaque type used by switch, which the kernel will then save in the interrupted thread struct.

Note that while SMP requires `CONFIG_USE_SWITCH`, the reverse is not true. A uniprocessor architecture built with `CONFIG_SMP` set to No might still decide to implement its context switching using *arch_switch()*.

API Reference

group `spinlock_apis`

Spinlock APIs.

Typedefs

typedef struct z_spinlock_key `k_spinlock_key_t`

Spinlock key type.

This type defines a "key" value used by a spinlock implementation to store the system interrupt state at the time of a call to *k_spin_lock()*. It is expected to be passed to a matching *k_spin_unlock()*.

This type is opaque and should not be inspected by application code.

Functions

ALWAYS_INLINE static *k_spinlock_key_t* `k_spin_lock`(struct *k_spinlock* *l)

Lock a spinlock.

This routine locks the specified spinlock, returning a key handle representing interrupt state needed at unlock time. Upon returning, the calling thread is guaranteed not to be suspended or interrupted on its current CPU until it calls *k_spin_unlock()*. The implementation guarantees mutual exclusion: exactly one thread on one CPU will return from *k_spin_lock()* at a time. Other CPUs trying to acquire a lock already held by another CPU will enter an implementation-defined busy loop ("spinning") until the lock is released.

Separate spin locks may be nested. It is legal to lock an (unlocked) spin lock while holding a different lock. Spin locks are not recursive, however: an attempt to acquire a spin lock that the CPU already holds will deadlock.

In circumstances where only one CPU exists, the behavior of *k_spin_lock()* remains as specified above, though obviously no spinning will take place. Implementations may be free to optimize in uniprocessor contexts such that the locking reduces to an interrupt mask operation.

Parameters

- `l` – A pointer to the spinlock to lock

Returns A key value that must be passed to *k_spin_unlock()* when the lock is released.

ALWAYS_INLINE static void `k_spin_unlock`(struct *k_spinlock* *l, *k_spinlock_key_t* key)

Unlock a spin lock.

This releases a lock acquired by *k_spin_lock()*. After this function is called, any CPU will be able to acquire the lock. If other CPUs are currently spinning inside *k_spin_lock()* waiting for this lock, exactly one of them will return synchronously with the lock held.

Spin locks must be properly nested. A call to *k_spin_unlock()* must be made on the lock object most recently locked using *k_spin_lock()*, using the key value that it returned. Attempts to unlock mis-nested locks, or to unlock locks that are not held, or to passing a key parameter other than the one returned from *k_spin_lock()*, are illegal. When CONFIG_SPIN_VALIDATE is set, some of these errors can be detected by the framework.

Parameters

- `l` – A pointer to the spinlock to release
- `key` – The value returned from *k_spin_lock()* when this lock was acquired

ALWAYS_INLINE static void `k_spin_release`(struct *k_spinlock* *l)

struct `k_spinlock`

#include <spinlock.h> Kernel Spin Lock.

This struct defines a spin lock record on which CPUs can wait with *k_spin_lock()*. Any number of spinlocks may be defined in application code.

7.13.2 Data Passing

These pages cover kernel objects which can be used to pass data between threads and ISRs.

The following table summarizes their high-level features.

Object	Bidirectional?	Data structure	Data item size	Data Alignment	ISRs can receive?	ISRs can send?	Overrun handling
FIFO	No	Queue	Arbitrary [1]	4 B [2]	Yes [3]	Yes	N/A
LIFO	No	Queue	Arbitrary [1]	4 B [2]	Yes [3]	Yes	N/A
Stack	No	Array	Word	Word	Yes [3]	Yes	Undefined behavior
Message queue	No	Ring buffer	Power of two	Power of two	Yes [3]	Yes	Pend thread or return -errno
Mailbox	Yes	Queue	Arbitrary [1]	Arbitrary	No	No	N/A
Pipe	No	Ring buffer [4]	Arbitrary	Arbitrary	No	No	Pend thread or return -errno

[1] Callers allocate space for queue overhead in the data elements themselves.

[2] Objects added with k_fifo_alloc_put() and k_lifo_alloc_put() do not have alignment constraints, but use temporary memory from the calling thread's resource pool.

[3] ISRs can receive only when passing K_NO_WAIT as the timeout argument.

[4] Optional.

Queues

A Queue in Zephyr is a kernel object that implements a traditional queue, allowing threads and ISRs to add and remove data items of any size. The queue is similar to a FIFO and serves as the underlying implementation for both *k_fifo* and *k_lifo*. For more information on usage see *k_fifo*.

Configuration Options Related configuration options:

- None

API Reference

group `queue_apis`

Defines

`K_QUEUE_DEFINE(name)`

Statically define and initialize a queue.

The queue can be accessed outside the module where it is defined using:

```
extern struct k_queue <name>;
```

Parameters

- `name` – Name of the queue.

Functions

void `k_queue_init(struct k_queue *queue)`

Initialize a queue.

This routine initializes a queue object, prior to its first use.

Parameters

- `queue` – Address of the queue.

Returns N/A

void `k_queue_cancel_wait(struct k_queue *queue)`

Cancel waiting on a queue.

This routine causes first thread pending on *queue*, if any, to return from *k_queue_get()* call with NULL value (as if timeout expired). If the queue is being waited on by *k_poll()*, it will return with -EINTR and K_POLL_STATE_CANCELLED state (and per above, subsequent *k_queue_get()* will return NULL).

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – Address of the queue.

Returns N/A

void k_queue_append(struct k_queue *queue, void *data)

Append an element to the end of a queue.

This routine appends a data item to *queue*. A queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Parameters

- queue – Address of the queue.
- data – Address of the data item.

Returns N/A

int32_t k_queue_alloc_append(struct k_queue *queue, void *data)

Append an element to a queue.

This routine appends a data item to *queue*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete) *isr-ok*

Parameters

- queue – Address of the queue.
- data – Address of the data item.

Return values

- 0 – on success
- -ENOMEM – if there isn't sufficient RAM in the caller's resource pool

void k_queue_prepend(struct k_queue *queue, void *data)

Prepend an element to a queue.

This routine prepends a data item to *queue*. A queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Parameters

- queue – Address of the queue.
- data – Address of the data item.

Returns N/A

int32_t k_queue_alloc_prepend(struct k_queue *queue, void *data)

Prepend an element to a queue.

This routine prepends a data item to *queue*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – Address of the queue.
- `data` – Address of the data item.

Return values

- `0` – on success
- `-ENOMEM` – if there isn't sufficient RAM in the caller's resource pool

void `k_queue_insert`(struct k_queue *queue, void *prev, void *data)

Inserts an element to a queue.

This routine inserts a data item to *queue* after previous item. A queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – Address of the queue.
- `prev` – Address of the previous data item.
- `data` – Address of the data item.

Returns N/A

int `k_queue_append_list`(struct k_queue *queue, void *head, void *tail)

Atomically append a list of elements to a queue.

This routine adds a list of data items to *queue* in one operation. The data items must be in a singly-linked list, with the first word in each data item pointing to the next data item; the list must be NULL-terminated.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – Address of the queue.
- `head` – Pointer to first node in singly-linked list.
- `tail` – Pointer to last node in singly-linked list.

Return values

- `0` – on success
- `-EINVAL` – on invalid supplied data

int `k_queue_merge_slist`(struct k_queue *queue, sys_slist_t *list)

Atomically add a list of elements to a queue.

This routine adds a list of data items to *queue* in one operation. The data items must be in a singly-linked list implemented using a sys_slist_t object. Upon completion, the original list is empty.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – Address of the queue.
- `list` – Pointer to sys_slist_t object.

Return values

- `0` – on success
- `-EINVAL` – on invalid data

void *`k_queue_get`(struct k_queue *queue, *k_timeout_t* timeout)

Get an element from a queue.

This routine removes first data item from *queue*. The first word of the data item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `queue` – Address of the queue.
- `timeout` – Non-negative waiting period to obtain a data item or one of the special values K_NO_WAIT and K_FOREVER.

Returns Address of the data item if successful; NULL if returned without waiting, or waiting period timed out.

bool `k_queue_remove`(struct k_queue *queue, void *data)

Remove an element from a queue.

This routine removes data item from *queue*. The first word of the data item is reserved for the kernel's use. Removing elements from k_queue rely on sys_slist_find_and_remove which is not a constant time operation.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `queue` – Address of the queue.
- `data` – Address of the data item.

Returns true if data item was removed

bool `k_queue_unique_append`(struct k_queue *queue, void *data)

Append an element to a queue only if it's not present already.

This routine appends data item to *queue*. The first word of the data item is reserved for the kernel's use. Appending elements to k_queue relies on sys_slist_is_node_in_list which is not a constant time operation.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – Address of the queue.
- `data` – Address of the data item.

Returns true if data item was added, false if not

int `k_queue_is_empty`(struct k_queue *queue)

Query a queue to see if it has data available.

Note that the data might be already gone by the time this function returns if other threads are also trying to read from the queue.

Function properties (list may not be complete) *isr-ok*

Parameters

- `queue` – Address of the queue.

Returns Non-zero if the queue is empty.

Returns 0 if data is available.

void *`k_queue_peek_head`(struct k_queue *queue)

Peek element at the head of queue.

Return element from the head of queue without removing it.

Parameters

- `queue` – Address of the queue.

Returns Head element, or NULL if queue is empty.

void *`k_queue_peek_tail`(struct k_queue *queue)

Peek element at the tail of queue.

Return element from the tail of queue without removing it.

Parameters

- `queue` – Address of the queue.

Returns Tail element, or NULL if queue is empty.

FIFOs

A *FIFO* is a kernel object that implements a traditional first in, first out (FIFO) queue, allowing threads and ISRs to add and remove data items of any size.

Concepts Any number of FIFOs can be defined (limited only by available RAM). Each FIFO is referenced by its memory address.

A FIFO has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A FIFO must be initialized before it can be used. This sets its queue to empty.

FIFO data items must be aligned on a word boundary, as the kernel reserves the first word of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 (or N+8) bytes of memory. There are no alignment or reserved space requirements for data items if they are added with *k_fifo_alloc_put()*, instead additional memory is temporarily allocated from the calling thread's resource pool.

A data item may be **added** to a FIFO by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the FIFO's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a FIFO by a thread. If the FIFO's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty FIFO simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

Note: The kernel does allow an ISR to remove an item from a FIFO, however the ISR must not attempt to wait if the FIFO is empty.

If desired, **multiple data items** can be added to a FIFO in a single operation if they are chained together into a singly-linked list. This capability can be useful if multiple writers are adding sets of related data items to the FIFO, as it ensures the data items in each set are not interleaved with other data items. Adding multiple data items to a FIFO is also more efficient than adding them one at a time, and can be used to guarantee that anyone who removes the first data item in a set will be able to remove the remaining data items without waiting.

Implementation

Defining a FIFO A FIFO is defined using a variable of type `k_fifo`. It must then be initialized by calling *k_fifo_init()*.

The following code defines and initializes an empty FIFO.

```
struct k_fifo my_fifo;

k_fifo_init(&my_fifo);
```

Alternatively, an empty FIFO can be defined and initialized at compile time by calling *K_FIFO_DEFINE*.

The following code has the same effect as the code segment above.

```
K_FIFO_DEFINE(my_fifo);
```

Writing to a FIFO A data item is added to a FIFO by calling *k_fifo_put()*.

The following code builds on the example above, and uses the FIFO to send data to one or more consumer threads.

```
struct data_item_t {
    void *fifo_reserved;   /* 1st word reserved for use by FIFO */
    ...
```

(continues on next page)

(continued from previous page)

```
};

struct data_item_t tx_data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_fifo_put(&my_fifo, &tx_data);

        ...
    }
}
```

Additionally, a singly-linked list of data items can be added to a FIFO by calling *k_fifo_put_list()* or *k_fifo_put_slist()*.

Finally, a data item can be added to a FIFO with *k_fifo_alloc_put()*. With this API, there is no need to reserve space for the kernel's use in the data item, instead additional memory will be allocated from the calling thread's resource pool until the item is read.

Reading from a FIFO A data item is removed from a FIFO by calling *k_fifo_get()*.

The following code builds on the example above, and uses the FIFO to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_fifo_get(&my_fifo, K_FOREVER);

        /* process FIFO data item */
        ...
    }
}
```

Suggested Uses Use a FIFO to asynchronously transfer data items of arbitrary size in a "first in, first out" manner.

Configuration Options Related configuration options:

- None

API Reference

group `fifo_apis`

Defines

`k_fifo_init`(fifo)

Initialize a FIFO queue.

This routine initializes a FIFO queue, prior to its first use.

Parameters

- `fifo` – Address of the FIFO queue.

Returns N/A

`k_fifo_cancel_wait`(fifo)

Cancel waiting on a FIFO queue.

This routine causes first thread pending on *fifo*, if any, to return from *k_fifo_get()* call with NULL value (as if timeout expired).

Function properties (list may not be complete) *isr-ok*

Parameters

- `fifo` – Address of the FIFO queue.

Returns N/A

`k_fifo_put`(fifo, data)

Add an element to a FIFO queue.

This routine adds a data item to *fifo*. A FIFO data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Parameters

- `fifo` – Address of the FIFO.
- `data` – Address of the data item.

Returns N/A

`k_fifo_alloc_put`(fifo, data)

Add an element to a FIFO queue.

This routine adds a data item to *fifo*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete) *isr-ok*

Parameters

- `fifo` – Address of the FIFO.
- `data` – Address of the data item.

Return values

- `0` – on success
- `-ENOMEM` – if there isn't sufficient RAM in the caller's resource pool

`k_fifo_put_list`(fifo, head, tail)

Atomically add a list of elements to a FIFO.

This routine adds a list of data items to *fifo* in one operation. The data items must be in a singly-linked list, with the first word of each data item pointing to the next data item; the list must be NULL-terminated.

Function properties (list may not be complete) *isr-ok*

Parameters

- `fifo` – Address of the FIFO queue.
- `head` – Pointer to first node in singly-linked list.
- `tail` – Pointer to last node in singly-linked list.

Returns N/A

`k_fifo_put_slist`(fifo, list)

Atomically add a list of elements to a FIFO queue.

This routine adds a list of data items to *fifo* in one operation. The data items must be in a singly-linked list implemented using a sys_slist_t object. Upon completion, the sys_slist_t object is invalid and must be re-initialized via *sys_slist_init()*.

Function properties (list may not be complete) *isr-ok*

Parameters

- `fifo` – Address of the FIFO queue.
- `list` – Pointer to sys_slist_t object.

Returns N/A

`k_fifo_get`(fifo, timeout)

Get an element from a FIFO queue.

This routine removes a data item from *fifo* in a "first in, first out" manner. The first word of the data item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `fifo` – Address of the FIFO queue.
- `timeout` – Waiting period to obtain a data item, or one of the special values K_NO_WAIT and K_FOREVER.

Returns Address of the data item if successful; NULL if returned without waiting, or waiting period timed out.

`k_fifo_is_empty(fifo)`

Query a FIFO queue to see if it has data available.

Note that the data might be already gone by the time this function returns if other threads is also trying to read from the FIFO.

Function properties (list may not be complete) *isr-ok*

Parameters

- `fifo` – Address of the FIFO queue.

Returns Non-zero if the FIFO queue is empty.

Returns 0 if data is available.

`k_fifo_peek_head(fifo)`

Peek element at the head of a FIFO queue.

Return element from the head of FIFO queue without removing it. A usecase for this is if elements of the FIFO object are themselves containers. Then on each iteration of processing, a head container will be peeked, and some data processed out of it, and only if the container is empty, it will be completely remove from the FIFO queue.

Parameters

- `fifo` – Address of the FIFO queue.

Returns Head element, or NULL if the FIFO queue is empty.

`k_fifo_peek_tail(fifo)`

Peek element at the tail of FIFO queue.

Return element from the tail of FIFO queue (without removing it). A usecase for this is if elements of the FIFO queue are themselves containers. Then it may be useful to add more data to the last container in a FIFO queue.

Parameters

- `fifo` – Address of the FIFO queue.

Returns Tail element, or NULL if a FIFO queue is empty.

`K_FIFO_DEFINE(name)`

Statically define and initialize a FIFO queue.

The FIFO queue can be accessed outside the module where it is defined using:

```
extern struct k_fifo <name>;
```

Parameters

- `name` – Name of the FIFO queue.

LIFOs

A *LIFO* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove data items of any size.

Concepts Any number of LIFOs can be defined (limited only by available RAM). Each LIFO is referenced by its memory address.

A LIFO has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A LIFO must be initialized before it can be used. This sets its queue to empty.

LIFO data items must be aligned on a word boundary, as the kernel reserves the first word of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 (or N+8) bytes of memory. There are no alignment or reserved space requirements for data items if they are added with *k_lifo_alloc_put()*, instead additional memory is temporarily allocated from the calling thread's resource pool.

A data item may be **added** to a LIFO by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the LIFO's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a LIFO by a thread. If the LIFO's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty LIFO simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

Note: The kernel does allow an ISR to remove an item from a LIFO, however the ISR must not attempt to wait if the LIFO is empty.

Implementation

Defining a LIFO A LIFO is defined using a variable of type `k_lifo`. It must then be initialized by calling *k_lifo_init()*.

The following defines and initializes an empty LIFO.

```
struct k_lifo my_lifo;

k_lifo_init(&my_lifo);
```

Alternatively, an empty LIFO can be defined and initialized at compile time by calling *K_LIFO_DEFINE*.

The following code has the same effect as the code segment above.

```
K_LIFO_DEFINE(my_lifo);
```

Writing to a LIFO A data item is added to a LIFO by calling *k_lifo_put()*.

The following code builds on the example above, and uses the LIFO to send data to one or more consumer threads.

```
struct data_item_t {
    void *LIFO_reserved;   /* 1st word reserved for use by LIFO */
    ...
};

struct data_item_t tx data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_lifo_put(&my_lifo, &tx_data);

        ...
    }
}
```

A data item can be added to a LIFO with *k_lifo_alloc_put()*. With this API, there is no need to reserve space for the kernel's use in the data item, instead additional memory will be allocated from the calling thread's resource pool until the item is read.

Reading from a LIFO A data item is removed from a LIFO by calling *k_lifo_get()*.

The following code builds on the example above, and uses the LIFO to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_lifo_get(&my_lifo, K_FOREVER);

        /* process LIFO data item */
        ...
    }
}
```

Suggested Uses Use a LIFO to asynchronously transfer data items of arbitrary size in a "last in, first out" manner.

Configuration Options Related configuration options:

- None.

API Reference

group `lifo_apis`

Defines

`k_lifo_init`(lifo)

Initialize a LIFO queue.

This routine initializes a LIFO queue object, prior to its first use.

Parameters

- `lifo` – Address of the LIFO queue.

Returns N/A

`k_lifo_put`(lifo, data)

Add an element to a LIFO queue.

This routine adds a data item to *lifo*. A LIFO queue data item must be aligned on a word boundary, and the first word of the item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Parameters

- `lifo` – Address of the LIFO queue.
- `data` – Address of the data item.

Returns N/A

`k_lifo_alloc_put`(lifo, data)

Add an element to a LIFO queue.

This routine adds a data item to *lifo*. There is an implicit memory allocation to create an additional temporary bookkeeping data structure from the calling thread's resource pool, which is automatically freed when the item is removed. The data itself is not copied.

Function properties (list may not be complete) *isr-ok*

Parameters

- `lifo` – Address of the LIFO.
- `data` – Address of the data item.

Return values

- `0` – on success
- `-ENOMEM` – if there isn't sufficient RAM in the caller's resource pool

`k_lifo_get`(lifo, timeout)

Get an element from a LIFO queue.

This routine removes a data item from *LIFO* in a "last in, first out" manner. The first word of the data item is reserved for the kernel's use.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `lifo` – Address of the LIFO queue.
- `timeout` – Waiting period to obtain a data item, or one of the special values K_NO_WAIT and K_FOREVER.

Returns Address of the data item if successful; NULL if returned without waiting, or waiting period timed out.

`K_LIFO_DEFINE(name)`

Statically define and initialize a LIFO queue.

The LIFO queue can be accessed outside the module where it is defined using:

```
extern struct k_lifo <name>;
```

Parameters

- `name` – Name of the fifo.

Stacks

A *stack* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove a limited number of integer data values.

Concepts Any number of stacks can be defined (limited only by available RAM). Each stack is referenced by its memory address.

A stack has the following key properties:

- A **queue** of integer data values that have been added but not yet removed. The queue is implemented using an array of stack_data_t values and must be aligned on a native word boundary. The stack_data_t type corresponds to the native word size i.e. 32 bits or 64 bits depending on the CPU architecture and compilation mode.
- A **maximum quantity** of data values that can be queued in the array.

A stack must be initialized before it can be used. This sets its queue to empty.

A data value can be **added** to a stack by a thread or an ISR. The value is given directly to a waiting thread, if one exists; otherwise the value is added to the LIFO's queue.

Note: If `CONFIG_NO_RUNTIME_CHECKS` is enabled, the kernel will *not* detect and prevent attempts to add a data value to a stack that has already reached its maximum quantity of queued values. Adding a data value to a stack that is already full will result in array overflow, and lead to unpredictable behavior.

A data value may be **removed** from a stack by a thread. If the stack's queue is empty a thread may choose to wait for it to be given. Any number of threads may wait on an empty stack simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

Note: The kernel does allow an ISR to remove an item from a stack, however the ISR must not attempt to wait if the stack is empty.

Implementation

Defining a Stack A stack is defined using a variable of type `k_stack`. It must then be initialized by calling *k_stack_init()* or *k_stack_alloc_init()*. In the latter case, a buffer is not provided and it is instead allocated from the calling thread's resource pool.

The following code defines and initializes an empty stack capable of holding up to ten word-sized data values.

```
#define MAX_ITEMS 10

stack_data_t my_stack_array[MAX_ITEMS];
struct k_stack my_stack;

k_stack_init(&my_stack, my_stack_array, MAX_ITEMS);
```

Alternatively, a stack can be defined and initialized at compile time by calling *K_STACK_DEFINE*.

The following code has the same effect as the code segment above. Observe that the macro defines both the stack and its array of data values.

```
K_STACK_DEFINE(my_stack, MAX_ITEMS);
```

Pushing to a Stack A data item is added to a stack by calling *k_stack_push()*.

The following code builds on the example above, and shows how a thread can create a pool of data structures by saving their memory addresses in a stack.

```
/* define array of data structures */
struct my_buffer_type {
    int field1;
    ...
    };
struct my_buffer_type my_buffers[MAX_ITEMS];

/* save address of each data structure in a stack */
for (int i = 0; i < MAX_ITEMS; i++) {
    k_stack_push(&my_stack, (stack_data_t)&my_buffers[i]);
}
```

Popping from a Stack A data item is taken from a stack by calling *k_stack_pop()*.

The following code builds on the example above, and shows how a thread can dynamically allocate an unused data structure. When the data structure is no longer required, the thread must push its address back on the stack to allow the data structure to be reused.

```
struct my_buffer_type *new_buffer;

k_stack_pop(&buffer_stack, (stack_data_t *)&new_buffer, K_FOREVER);
new_buffer->field1 = ...
```

Suggested Uses Use a stack to store and retrieve integer data values in a "last in, first out" manner, when the maximum number of stored items is known.

Configuration Options Related configuration options:

- None.

API Reference

group `stack_apis`

Defines

`K_STACK_DEFINE(name, stack_num_entries)`

Statically define and initialize a stack.

The stack can be accessed outside the module where it is defined using:

```
extern struct k_stack <name>;
```

Parameters

- `name` – Name of the stack.
- `stack_num_entries` – Maximum number of values that can be stacked.

Functions

void `k_stack_init(struct k_stack *stack, stack_data_t *buffer, uint32_t num_entries)`

Initialize a stack.

This routine initializes a stack object, prior to its first use.

Parameters

- `stack` – Address of the stack.
- `buffer` – Address of array used to hold stacked values.
- `num_entries` – Maximum number of values that can be stacked.

Returns N/A

int32_t `k_stack_alloc_init(struct k_stack *stack, uint32_t num_entries)`

Initialize a stack.

This routine initializes a stack object, prior to its first use. Internal buffers will be allocated from the calling thread's resource pool. This memory will be released if *k_stack_cleanup()* is called, or userspace is enabled and the stack object loses all references to it.

Parameters

- `stack` – Address of the stack.

- `num_entries` – Maximum number of values that can be stacked.

Returns -ENOMEM if memory couldn't be allocated

int `k_stack_cleanup`(struct k_stack *stack)

Release a stack's allocated buffer.

If a stack object was given a dynamically allocated buffer via *k_stack_alloc_init()*, this will free it. This function does nothing if the buffer wasn't dynamically allocated.

Parameters

- `stack` – Address of the stack.

Return values

- `0` – on success
- `-EAGAIN` – when object is still in use

int `k_stack_push`(struct k_stack *stack, stack_data_t data)

Push an element onto a stack.

This routine adds a stack_data_t value *data* to *stack*.

Function properties (list may not be complete) *isr-ok*

Parameters

- `stack` – Address of the stack.
- `data` – Value to push onto the stack.

Return values

- `0` – on success
- `-ENOMEM` – if stack is full

int `k_stack_pop`(struct k_stack *stack, stack_data_t *data, *k_timeout_t* timeout)

Pop an element from a stack.

This routine removes a stack_data_t value from *stack* in a "last in,

first out" manner and stores the value in *data*.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- `stack` – Address of the stack.
- `data` – Address of area to hold the value popped from the stack.
- `timeout` – Waiting period to obtain a value, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Element popped from stack.
- `-EBUSY` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.

Message Queues

A *message queue* is a kernel object that implements a simple message queue, allowing threads and ISRs to asynchronously send and receive fixed-size data items.

Concepts Any number of message queues can be defined (limited only by available RAM). Each message queue is referenced by its memory address.

A message queue has the following key properties:

- A **ring buffer** of data items that have been sent but not yet received.
- A **data item size**, measured in bytes.
- A **maximum quantity** of data items that can be queued in the ring buffer.

The message queue's ring buffer must be aligned to an N-byte boundary, where N is a power of 2 (i.e. 1, 2, 4, 8, ...). To ensure that the messages stored in the ring buffer are similarly aligned to this boundary, the data item size must also be a multiple of N.

A message queue must be initialized before it can be used. This sets its ring buffer to empty.

A data item can be **sent** to a message queue by a thread or an ISR. The data item pointed at by the sending thread is copied to a waiting thread, if one exists; otherwise the item is copied to the message queue's ring buffer, if space is available. In either case, the size of the data area being sent *must* equal the message queue's data item size.

If a thread attempts to send a data item when the ring buffer is full, the sending thread may choose to wait for space to become available. Any number of sending threads may wait simultaneously when the ring buffer is full; when space becomes available it is given to the highest priority sending thread that has waited the longest.

A data item can be **received** from a message queue by a thread. The data item is copied to the area specified by the receiving thread; the size of the receiving area *must* equal the message queue's data item size.

If a thread attempts to receive a data item when the ring buffer is empty, the receiving thread may choose to wait for a data item to be sent. Any number of receiving threads may wait simultaneously when the ring buffer is empty; when a data item becomes available it is given to the highest priority receiving thread that has waited the longest.

A thread can also **peek** at the message on the head of a message queue without removing it from the queue. The data item is copied to the area specified by the receiving thread; the size of the receiving area *must* equal the message queue's data item size.

Note: The kernel does allow an ISR to receive an item from a message queue, however the ISR must not attempt to wait if the message queue is empty.

Implementation

Defining a Message Queue A message queue is defined using a variable of type *k_msgq*. It must then be initialized by calling *k_msgq_init()*.

The following code defines and initializes an empty message queue that is capable of holding 10 items, each of which is 12 bytes long.

```
struct data_item_type {
    uint32_t field1;
    uint32_t field2;
    uint32_t field3;
};

char __aligned(4) my_msgq_buffer[10 * sizeof(struct data_item_type)];
struct k_msgq my_msgq;

k_msgq_init(&my_msgq, my_msgq_buffer, sizeof(struct data_item_type), 10);
```

Alternatively, a message queue can be defined and initialized at compile time by calling *K_MSGQ_DEFINE*.

The following code has the same effect as the code segment above. Observe that the macro defines both the message queue and its buffer.

```
K_MSGQ_DEFINE(my_msgq, sizeof(struct data_item_type), 10, 4);
```

The following code demonstrates an alignment implementation for the structure defined in the previous example code. `aligned` means each `data_item_type` will begin on the specified byte boundary. `aligned(4)` means that the structure is aligned to an address that is divisible by 4.

```
typedef struct {
    uint32_t field1;
    uint32_t field2;
    uint32_t field3;
}__attribute__((aligned(4))) data_item_type;
```

Writing to a Message Queue A data item is added to a message queue by calling *k_msgq_put()*.

The following code builds on the example above, and uses the message queue to pass data items from a producing thread to one or more consuming threads. If the message queue fills up because the consumers can't keep up, the producing thread throws away all existing data so the newer data can be saved.

```
void producer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* create data item to send (e.g. measurement, timestamp, ...) */
        data = ...

        /* send data to consumers */
        while (k_msgq_put(&my_msgq, &data, K_NO_WAIT) != 0) {
```

(continues on next page)

(continued from previous page)

```
            /* message queue is full: purge old data & try again */
            k_msgq_purge(&my_msgq);
        }

        /* data item was successfully added to message queue */
    }
}
```

Reading from a Message Queue A data item is taken from a message queue by calling *k_msgq_get()*.

The following code builds on the example above, and uses the message queue to process data items generated by one or more producing threads. Note that the return value of *k_msgq_get()* should be tested as -ENOMSG can be returned due to *k_msgq_purge()*.

```
void consumer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* get a data item */
        k_msgq_get(&my_msgq, &data, K_FOREVER);

        /* process data item */
        ...
    }
}
```

Peeking into a Message Queue A data item is read from a message queue by calling *k_msgq_peek()*.

The following code peeks into the message queue to read the data item at the head of the queue that is generated by one or more producing threads.

```
void consumer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* read a data item by peeking into the queue */
        k_msgq_peek(&my_msgq, &data);

        /* process data item */
        ...
    }
}
```

Suggested Uses Use a message queue to transfer small data items between threads in an asynchronous manner.

Note: A message queue can be used to transfer large data items, if desired. However, this can increase interrupt latency as interrupts are locked while a data item is written or read. The time to write or read a data item increases linearly with its size since the item is copied in its entirety to or from the buffer in memory. For this reason, it is usually preferable to transfer large data items by exchanging a pointer to the data item, rather than the data item itself.

A synchronous transfer can be achieved by using the kernel's mailbox object type.

Configuration Options Related configuration options:

- None.

API Reference

group `msgq_apis`

Defines

`K_MSGQ_FLAG_ALLOC`

`K_MSGQ_DEFINE(q_name, q_msg_size, q_max_msgs, q_align)`

Statically define and initialize a message queue.

The message queue's ring buffer contains space for *q_max_msgs* messages, each of which is *q_msg_size* bytes long. The buffer is aligned to a *q_align* -byte boundary, which must be a power of 2. To ensure that each message is similarly aligned to this boundary, *q_msg_size* must also be a multiple of *q_align*.

The message queue can be accessed outside the module where it is defined using:

```
extern struct k_msgq <name>;
```

Parameters

- `q_name` – Name of the message queue.
- `q_msg_size` – Message size (in bytes).
- `q_max_msgs` – Maximum number of messages that can be queued.
- `q_align` – Alignment of the message queue's ring buffer.

Functions

void `k_msgq_init`(struct *k_msgq* *msgq, char *buffer, size_t msg_size, uint32_t max_msgs)

Initialize a message queue.

This routine initializes a message queue object, prior to its first use.

The message queue's ring buffer must contain space for *max_msgs* messages, each of which is *msg_size* bytes long. The buffer must be aligned to an N-byte boundary, where N is a power of 2 (i.e. 1, 2, 4, ...). To ensure that each message is similarly aligned to this boundary, *q_msg_size* must also be a multiple of N.

Parameters

- `msgq` – Address of the message queue.
- `buffer` – Pointer to ring buffer that holds queued messages.
- `msg_size` – Message size (in bytes).
- `max_msgs` – Maximum number of messages that can be queued.

Returns N/A

int `k_msgq_alloc_init`(struct *k_msgq* *msgq, size_t msg_size, uint32_t max_msgs)

Initialize a message queue.

This routine initializes a message queue object, prior to its first use, allocating its internal ring buffer from the calling thread's resource pool.

Memory allocated for the ring buffer can be released by calling *k_msgq_cleanup()*, or if userspace is enabled and the msgq object loses all of its references.

Parameters

- `msgq` – Address of the message queue.
- `msg_size` – Message size (in bytes).
- `max_msgs` – Maximum number of messages that can be queued.

Returns 0 on success, -ENOMEM if there was insufficient memory in the thread's resource pool, or -EINVAL if the size parameters cause an integer overflow.

int `k_msgq_cleanup`(struct *k_msgq* *msgq)

Release allocated buffer for a queue.

Releases memory allocated for the ring buffer.

Parameters

- `msgq` – message queue to cleanup

Return values

- `0` – on success
- `-EBUSY` – Queue not empty

int `k_msgq_put`(struct *k_msgq* *msgq, const void *data, *k_timeout_t* timeout)

Send a message to a message queue.

This routine sends a message to message queue *q*.

Function properties (list may not be complete) *isr-ok*

Note: The message content is copied from *data* into *msgq* and the *data* pointer is not retained, so the message content will not be modified by this function.

Parameters

- `msgq` – Address of the message queue.
- `data` – Pointer to the message.
- `timeout` – Non-negative waiting period to add the message, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – Message sent.
- `-ENOMSG` – Returned without waiting or queue purged.
- `-EAGAIN` – Waiting period timed out.

int k_msgq_get(struct *k_msgq* *msgq, void *data, *k_timeout_t* timeout)

Receive a message from a message queue.

This routine receives a message from message queue *q* in a "first in,

first out" manner.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Parameters

- msgq – Address of the message queue.
- data – Address of area to hold the received message.
- timeout – Waiting period to receive the message, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- 0 – Message received.
- -ENOMSG – Returned without waiting.
- -EAGAIN – Waiting period timed out.

int k_msgq_peek(struct *k_msgq* *msgq, void *data)

Peek/read a message from a message queue.

This routine reads a message from message queue *q* in a "first in,

first out" manner and leaves the message in the queue.

Function properties (list may not be complete) *isr-ok*

Parameters

- msgq – Address of the message queue.
- data – Address of area to hold the message read from the queue.

Return values

- 0 – Message read.
- -ENOMSG – Returned when the queue has no message.

void k_msgq_purge(struct *k_msgq* *msgq)

Purge a message queue.

This routine discards all unreceived messages in a message queue's ring buffer. Any threads that are blocked waiting to send a message to the message queue are unblocked and see an -ENOMSG error code.

Parameters

- msgq – Address of the message queue.

Returns N/A

uint32_t `k_msgq_num_free_get`(struct *k_msgq* *msgq)

Get the amount of free space in a message queue.

This routine returns the number of unused entries in a message queue's ring buffer.

Parameters

- `msgq` – Address of the message queue.

Returns Number of unused ring buffer entries.

void `k_msgq_get_attrs`(struct *k_msgq* *msgq, struct *k_msgq_attrs* *attrs)

Get basic attributes of a message queue.

This routine fetches basic attributes of message queue into attr argument.

Parameters

- `msgq` – Address of the message queue.
- `attrs` – pointer to message queue attribute structure.

Returns N/A

uint32_t `k_msgq_num_used_get`(struct *k_msgq* *msgq)

Get the number of messages in a message queue.

This routine returns the number of messages in a message queue's ring buffer.

Parameters

- `msgq` – Address of the message queue.

Returns Number of messages.

struct `k_msgq`

#include <kernel.h> Message Queue Structure.

Public Members

_wait_q_t `wait_q`

Message queue wait queue

struct *k_spinlock* `lock`

Lock

size_t `msg_size`

Message size

uint32_t `max_msgs`

Maximal number of messages

char *`buffer_start`

Start of message buffer

char *`buffer_end`

End of message buffer

char *read_ptr
: Read pointer

char *write_ptr
: Write pointer

uint32_t used_msgs
: Number of used messages

uint8_t flags
: Message queue

struct k_msgq_attrs
: *#include <kernel.h>* Message Queue Attributes.

Public Members

size_t msg_size
: Message Size

uint32_t max_msgs
: Maximal number of messages

uint32_t used_msgs
: Used messages

Mailboxes

A *mailbox* is a kernel object that provides enhanced message queue capabilities that go beyond the capabilities of a message queue object. A mailbox allows threads to send and receive messages of any size synchronously or asynchronously.

- *API Reference*

Concepts Any number of mailboxes can be defined (limited only by available RAM). Each mailbox is referenced by its memory address.

A mailbox has the following key properties:

- A **send queue** of messages that have been sent but not yet received.
- A **receive queue** of threads that are waiting to receive a message.

A mailbox must be initialized before it can be used. This sets both of its queues to empty.

A mailbox allows threads, but not ISRs, to exchange messages. A thread that sends a message is known as the **sending thread**, while a thread that receives the message is known as the **receiving thread**. Each message may be received by only one thread (i.e. point-to-multipoint and broadcast messaging is not supported).

Messages exchanged using a mailbox are handled non-anonymously, allowing both threads participating in an exchange to know (and even specify) the identity of the other thread.

Message Format A **message descriptor** is a data structure that specifies where a message's data is located, and how the message is to be handled by the mailbox. Both the sending thread and the receiving thread supply a message descriptor when accessing a mailbox. The mailbox uses the message descriptors to perform a message exchange between compatible sending and receiving threads. The mailbox also updates certain message descriptor fields during the exchange, allowing both threads to know what has occurred.

A mailbox message contains zero or more bytes of **message data**. The size and format of the message data is application-defined, and can vary from one message to the next. There are two forms of message data:

- A **message buffer** is an area of memory provided by the thread that sends or receives the message. An array or structure variable can often be used for this purpose.
- A **message block** is an area of memory allocated from a memory pool.

A message may *not* have both a message buffer and a message block. A message that has neither form of message data is called an **empty message**.

Note: A message whose message buffer or memory block exists, but contains zero bytes of actual data, is *not* an empty message.

Message Lifecycle The life cycle of a message is straightforward. A message is created when it is given to a mailbox by the sending thread. The message is then owned by the mailbox until it is given to a receiving thread. The receiving thread may retrieve the message data when it receives the message from the mailbox, or it may perform data retrieval during a second, subsequent mailbox operation. Only when data retrieval has occurred is the message deleted by the mailbox.

Thread Compatibility A sending thread can specify the address of the thread to which the message is sent, or send it to any thread by specifying `K_ANY`. Likewise, a receiving thread can specify the address of the thread from which it wishes to receive a message, or it can receive a message from any thread by specifying `K_ANY`. A message is exchanged only when the requirements of both the sending thread and receiving thread are satisfied; such threads are said to be **compatible**.

For example, if thread A sends a message to thread B (and only thread B) it will be received by thread B if thread B tries to receive a message from thread A or if thread B tries to receive from any thread. The

exchange will not occur if thread B tries to receive a message from thread C. The message can never be received by thread C, even if it tries to receive a message from thread A (or from any thread).

Message Flow Control Mailbox messages can be exchanged **synchronously** or **asynchronously**. In a synchronous exchange, the sending thread blocks until the message has been fully processed by the receiving thread. In an asynchronous exchange, the sending thread does not wait until the message has been received by another thread before continuing; this allows the sending thread to do other work (such as gather data that will be used in the next message) *before* the message is given to a receiving thread and fully processed. The technique used for a given message exchange is determined by the sending thread.

The synchronous exchange technique provides an implicit form of flow control, preventing a sending thread from generating messages faster than they can be consumed by receiving threads. The asynchronous exchange technique provides an explicit form of flow control, which allows a sending thread to determine if a previously sent message still exists before sending a subsequent message.

Implementation

Defining a Mailbox A mailbox is defined using a variable of type *k_mbox*. It must then be initialized by calling *k_mbox_init()*.

The following code defines and initializes an empty mailbox.

```
struct k_mbox my_mailbox;

k_mbox_init(&my_mailbox);
```

Alternatively, a mailbox can be defined and initialized at compile time by calling *K_MBOX_DEFINE*.

The following code has the same effect as the code segment above.

```
K_MBOX_DEFINE(my_mailbox);
```

Message Descriptors A message descriptor is a structure of type *k_mbox_msg*. Only the fields listed below should be used; any other fields are for internal mailbox use only.

info A 32-bit value that is exchanged by the message sender and receiver, and whose meaning is defined by the application. This exchange is bi-directional, allowing the sender to pass a value to the receiver during any message exchange, and allowing the receiver to pass a value to the sender during a synchronous message exchange.

size The message data size, in bytes. Set it to zero when sending an empty message, or when sending a message buffer or message block with no actual data. When receiving a message, set it to the maximum amount of data desired, or to zero if the message data is not wanted. The mailbox updates this field with the actual number of data bytes exchanged once the message is received.

tx_data A pointer to the sending thread's message buffer. Set it to `NULL` when sending a memory block, or when sending an empty message. Leave this field uninitialized when receiving a message.

tx_block The descriptor for the sending thread's memory block. Set tx_block.data to `NULL` when sending an empty message. Leave this field uninitialized when sending a message buffer, or when receiving a message.

tx_target_thread The address of the desired receiving thread. Set it to `K_ANY` to allow any thread to receive the message. Leave this field uninitialized when receiving a message. The mailbox updates this field with the actual receiver's address once the message is received.

rx_source_thread The address of the desired sending thread. Set it to `K_ANY` to receive a message sent by any thread. Leave this field uninitialized when sending a message. The mailbox updates this field with the actual sender's address when the message is put into the mailbox.

Sending a Message A thread sends a message by first creating its message data, if any. A message buffer is typically used when the data volume is small, and the cost of copying the data is less than the cost of allocating and freeing a message block.

Next, the sending thread creates a message descriptor that characterizes the message to be sent, as described in the previous section.

Finally, the sending thread calls a mailbox send API to initiate the message exchange. The message is immediately given to a compatible receiving thread, if one is currently waiting. Otherwise, the message is added to the mailbox's send queue.

Any number of messages may exist simultaneously on a send queue. The messages in the send queue are sorted according to the priority of the sending thread. Messages of equal priority are sorted so that the oldest message can be received first.

For a synchronous send operation, the operation normally completes when a receiving thread has both received the message and retrieved the message data. If the message is not received before the waiting period specified by the sending thread is reached, the message is removed from the mailbox's send queue and the send operation fails. When a send operation completes successfully the sending thread can examine the message descriptor to determine which thread received the message, how much data was exchanged, and the application-defined info value supplied by the receiving thread.

Note: A synchronous send operation may block the sending thread indefinitely, even when the thread specifies a maximum waiting period. The waiting period only limits how long the mailbox waits before the message is received by another thread. Once a message is received there is *no* limit to the time the receiving thread may take to retrieve the message data and unblock the sending thread.

For an asynchronous send operation, the operation always completes immediately. This allows the sending thread to continue processing regardless of whether the message is given to a receiving thread immediately or added to the send queue. The sending thread may optionally specify a semaphore that the mailbox gives when the message is deleted by the mailbox, for example, when the message has been received and its data retrieved by a receiving thread. The use of a semaphore allows the sending thread to easily implement a flow control mechanism that ensures that the mailbox holds no more than an application-specified number of messages from a sending thread (or set of sending threads) at any point in time.

Note: A thread that sends a message asynchronously has no way to determine which thread received the message, how much data was exchanged, or the application-defined info value supplied by the receiving thread.

Sending an Empty Message This code uses a mailbox to synchronously pass 4 byte random values to any consuming thread that wants one. The message "info" field is large enough to carry the information being exchanged, so the data portion of the message isn't used.

```
void producer_thread(void)
{
    struct k_mbox_msg send_msg;

    while (1) {

        /* generate random value to send */
        uint32_t random_value = sys_rand32_get();

        /* prepare to send empty message */
        send_msg.info = random_value;
        send_msg.size = 0;
```

(continues on next page)

(continued from previous page)

```
        send_msg.tx_data = NULL;
        send_msg.tx_block.data = NULL;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);
    }
}
```

Sending Data Using a Message Buffer This code uses a mailbox to synchronously pass variable-sized requests from a producing thread to any consuming thread that wants it. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```
void producer_thread(void)
{
    char buffer[100];
    int buffer_bytes_used;

    struct k_mbox_msg send_msg;

    while (1) {

        /* generate data to send */
        ...
        buffer_bytes_used = ... ;
        memcpy(buffer, source, buffer_bytes_used);

        /* prepare to send message */
        send_msg.info = buffer_bytes_used;
        send_msg.size = buffer_bytes_used;
        send_msg.tx_data = buffer;
        send_msg.tx_block.data = NULL;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);

        /* info, size, and tx_target_thread fields have been updated */

        /* verify that message data was fully received */
        if (send_msg.size < buffer_bytes_used) {
            printf("some message data dropped during transfer!");
            printf("receiver only had room for %d bytes", send_msg.info);
        }
    }
}
```

Sending Data Using a Message Block This code uses a mailbox to send asynchronous messages. A semaphore is used to hold off the sending of a new message until the previous message has been consumed, so that a backlog of messages doesn't build up when the consuming thread is unable to keep up.

The message data is stored in a memory block obtained from a memory pool, thereby eliminating unneeded data copying when exchanging large messages. The memory pool contains only two blocks: one block gets filled with data while the previously sent block is being processed

```
/* define a semaphore, indicating that no message has been sent */
K_SEM_DEFINE(my_sem, 1, 1);

/* define a memory pool containing 2 blocks of 4096 bytes */
K_MEM_POOL_DEFINE(my_pool, 4096, 4096, 2, 4);

void producer_thread(void)
{
    struct k_mbox_msg send_msg;

    volatile char *hw_buffer;

    while (1) {
        /* allocate a memory block to hold the message data */
        k_mem_pool_alloc(&my_pool, &send_msg.tx_block, 4096, K_FOREVER);

        /* keep overwriting the hardware-generated data in the block    */
        /* until the previous message has been received by the consumer */
        do {
            memcpy(send_msg.tx_block.data, hw_buffer, 4096);
        } while (k_sem_take(&my_sem, K_NO_WAIT) != 0);

        /* finish preparing to send message */
        send_msg.size = 4096;
        send_msg.tx_target_thread = K_ANY;

        /* send message containing most current data and loop around */
        k_mbox_async_put(&my_mailbox, &send_msg, &my_sem);
    }
}
```

Receiving a Message A thread receives a message by first creating a message descriptor that characterizes the message it wants to receive. It then calls one of the mailbox receive APIs. The mailbox searches its send queue and takes the message from the first compatible thread it finds. If no compatible thread exists, the receiving thread may choose to wait for one. If no compatible thread appears before the waiting period specified by the receiving thread is reached, the receive operation fails. Once a receive operation completes successfully the receiving thread can examine the message descriptor to determine which thread sent the message, how much data was exchanged, and the application-defined info value supplied by the sending thread.

Any number of receiving threads may wait simultaneously on a mailboxes' receive queue. The threads are sorted according to their priority; threads of equal priority are sorted so that the one that started waiting first can receive a message first.

Note: Receiving threads do not always receive messages in a first in, first out (FIFO) order, due to the thread compatibility constraints specified by the message descriptors. For example, if thread A waits to receive a message only from thread X and then thread B waits to receive a message from thread Y, an incoming message from thread Y to any thread will be given to thread B and thread A will continue to wait.

The receiving thread controls both the quantity of data it retrieves from an incoming message and where the data ends up. The thread may choose to take all of the data in the message, to take only the initial part of the data, or to take no data at all. Similarly, the thread may choose to have the data copied into a message buffer of its choice or to have it placed in a message block. A message buffer is typically used when the volume of data involved is small, and the cost of copying the data is less than the cost of allocating and freeing a memory pool block.

The following sections outline various approaches a receiving thread may use when retrieving message data.

Retrieving Data at Receive Time The most straightforward way for a thread to retrieve message data is to specify a message buffer when the message is received. The thread indicates both the location of the message buffer (which must not be `NULL`) and its size.

The mailbox copies the message's data to the message buffer as part of the receive operation. If the message buffer is not big enough to contain all of the message's data, any uncopied data is lost. If the message is not big enough to fill all of the buffer with data, the unused portion of the message buffer is left unchanged. In all cases the mailbox updates the receiving thread's message descriptor to indicate how many data bytes were copied (if any).

The immediate data retrieval technique is best suited for small messages where the maximum size of a message is known in advance.

Note: This technique can be used when the message data is actually located in a memory block supplied by the sending thread. The mailbox copies the data into the message buffer specified by the receiving thread, then frees the message block back to its memory pool. This allows a receiving thread to retrieve message data without having to know whether the data was sent using a message buffer or a message block.

The following code uses a mailbox to process variable-sized requests from any producing thread, using the immediate data retrieval technique. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[100];

    int i;
    int total;

    while (1) {
        /* prepare to receive message */
        recv_msg.info = 100;
        recv_msg.size = 100;
        recv_msg.rx_source_thread = K_ANY;

        /* get a data item, waiting as long as needed */
        k_mbox_get(&my_mailbox, &recv_msg, buffer, K_FOREVER);

        /* info, size, and rx_source_thread fields have been updated */

        /* verify that message data was fully received */
        if (recv_msg.info != recv_msg.size) {
            printf("some message data dropped during transfer!");
            printf("sender tried to send %d bytes", recv_msg.info);
        }

        /* compute sum of all message bytes (from 0 to 100 of them) */
        total = 0;
        for (i = 0; i < recv_msg.size; i++) {
            total += buffer[i];
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Retrieving Data Later Using a Message Buffer A receiving thread may choose to defer message data retrieval at the time the message is received, so that it can retrieve the data into a message buffer at a later time. The thread does this by specifying a message buffer location of `NULL` and a size indicating the maximum amount of data it is willing to retrieve later.

The mailbox does not copy any message data as part of the receive operation. However, the mailbox still updates the receiving thread's message descriptor to indicate how many data bytes are available for retrieval.

The receiving thread must then respond as follows:

- If the message descriptor size is zero, then either the sender's message contained no data or the receiving thread did not want to receive any data. The receiving thread does not need to take any further action, since the mailbox has already completed data retrieval and deleted the message.
- If the message descriptor size is non-zero and the receiving thread still wants to retrieve the data, the thread must call *k_mbox_data_get()* and supply a message buffer large enough to hold the data. The mailbox copies the data into the message buffer and deletes the message.
- If the message descriptor size is non-zero and the receiving thread does *not* want to retrieve the data, the thread must call *k_mbox_data_get()* . and specify a message buffer of `NULL`. The mailbox deletes the message without copying the data.

The subsequent data retrieval technique is suitable for applications where immediate retrieval of message data is undesirable. For example, it can be used when memory limitations make it impractical for the receiving thread to always supply a message buffer capable of holding the largest possible incoming message.

Note: This technique can be used when the message data is actually located in a memory block supplied by the sending thread. The mailbox copies the data into the message buffer specified by the receiving thread, then frees the message block back to its memory pool. This allows a receiving thread to retrieve message data without having to know whether the data was sent using a message buffer or a message block.

The following code uses a mailbox's deferred data retrieval mechanism to get message data from a producing thread only if the message meets certain criteria, thereby eliminating unneeded data copying. The message "info" field supplied by the sender is used to classify the message.

```
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[10000];

    while (1) {
        /* prepare to receive message */
        recv_msg.size = 10000;
        recv_msg.rx_source_thread = K_ANY;

        /* get message, but not its data */
        k_mbox_get(&my_mailbox, &recv_msg, NULL, K_FOREVER);

        /* get message data for only certain types of messages */
        if (is_message_type_ok(recv_msg.info)) {
            /* retrieve message data and delete the message */
```

(continues on next page)

(continued from previous page)

```
            k_mbox_data_get(&recv_msg, buffer);

            /* process data in "buffer" */
            ...
        } else {
            /* ignore message data and delete the message */
            k_mbox_data_get(&recv_msg, NULL);
        }
    }
}
```

Retrieving Data Later Using a Message Block A receiving thread may choose to retrieve message data into a memory block, rather than a message buffer. This is done in much the same way as retrieving data subsequently into a message buffer — the receiving thread first receives the message without its data, then retrieves the data by calling `k_mbox_data_block_get()`. The mailbox fills in the block descriptor supplied by the receiving thread, allowing the thread to access the data. The mailbox also deletes the received message, since data retrieval has been completed. The receiving thread is then responsible for freeing the message block back to the memory pool when the data is no longer needed.

This technique is best suited for applications where the message data has been sent using a memory block.

Note: This technique can be used when the message data is located in a message buffer supplied by the sending thread. The mailbox automatically allocates a memory block and copies the message data into it. However, this is much less efficient than simply retrieving the data into a message buffer supplied by the receiving thread. In addition, the receiving thread must be designed to handle cases where the data retrieval operation fails because the mailbox cannot allocate a suitable message block from the memory pool. If such cases are possible, the receiving thread must either try retrieving the data at a later time or instruct the mailbox to delete the message without retrieving the data.

The following code uses a mailbox to receive messages sent using a memory block, thereby eliminating unneeded data copying when processing a large message. (The messages may be sent synchronously or asynchronously.)

```
/* define a memory pool containing 1 block of 10000 bytes */
K_MEM_POOL_DEFINE(my_pool, 10000, 10000, 1, 4);

void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    struct k_mem_block recv_block;

    int total;
    char *data_ptr;
    int i;

    while (1) {
        /* prepare to receive message */
        recv_msg.size = 10000;
        recv_msg.rx_source_thread = K_ANY;

        /* get message, but not its data */
        k_mbox_get(&my_mailbox, &recv_msg, NULL, K_FOREVER);

        /* get message data as a memory block and discard message */
```

(continues on next page)

(continued from previous page)

```
        k_mbox_data_block_get(&recv_msg, &my_pool, &recv_block, K_FOREVER);

        /* compute sum of all message bytes in memory block */
        total = 0;
        data_ptr = (char *)(recv_block.data);
        for (i = 0; i < recv_msg.size; i++) {
            total += data_ptr++;
        }

        /* release memory block containing data */
        k_mem_pool_free(&recv_block);
    }
}
```

Note: An incoming message that was sent using a message buffer is also processed correctly by this algorithm, since the mailbox automatically allocates a memory block from the memory pool and fills it with the message data. However, the performance benefit of using the memory block approach is lost.

Suggested Uses Use a mailbox to transfer data items between threads whenever the capabilities of a message queue are insufficient.

Configuration Options Related configuration options:

- `CONFIG_NUM_MBOX_ASYNC_MSGS`

API Reference

group `mailbox_apis`

Defines

`K_MBOX_DEFINE(name)`

Statically define and initialize a mailbox.

The mailbox is to be accessed outside the module where it is defined using:

```
extern struct k_mbox <name>;
```

Parameters

- `name` – Name of the mailbox.

Functions

void `k_mbox_init`(struct *k_mbox* *mbox)

Initialize a mailbox.

This routine initializes a mailbox object, prior to its first use.

Parameters

- `mbox` – Address of the mailbox.

Returns N/A

int k_mbox_put(struct *k_mbox* *mbox, struct *k_mbox_msg* *tx_msg, *k_timeout_t* timeout)

Send a mailbox message in a synchronous manner.

This routine sends a message to *mbox* and waits for a receiver to both receive and process it. The message data may be in a buffer, in a memory pool block, or non-existent (i.e. an empty message).

Parameters

- mbox – Address of the mailbox.
- tx_msg – Address of the transmit message descriptor.
- timeout – Waiting period for the message to be received, or one of the special values K_NO_WAIT and K_FOREVER. Once the message has been received, this routine waits as long as necessary for the message to be completely processed.

Return values

- 0 – Message sent.
- -ENOMSG – Returned without waiting.
- -EAGAIN – Waiting period timed out.

void k_mbox_async_put(struct *k_mbox* *mbox, struct *k_mbox_msg* *tx_msg, struct k_sem *sem)

Send a mailbox message in an asynchronous manner.

This routine sends a message to *mbox* without waiting for a receiver to process it. The message data may be in a buffer, in a memory pool block, or non-existent (i.e. an empty message). Optionally, the semaphore *sem* will be given when the message has been both received and completely processed by the receiver.

Parameters

- mbox – Address of the mailbox.
- tx_msg – Address of the transmit message descriptor.
- sem – Address of a semaphore, or NULL if none is needed.

Returns N/A

int k_mbox_get(struct *k_mbox* *mbox, struct *k_mbox_msg* *rx_msg, void *buffer, *k_timeout_t* timeout)

Receive a mailbox message.

This routine receives a message from *mbox*, then optionally retrieves its data and disposes of the message.

Parameters

- mbox – Address of the mailbox.
- rx_msg – Address of the receive message descriptor.
- buffer – Address of the buffer to receive data, or NULL to defer data retrieval and message disposal until later.
- timeout – Waiting period for a message to be received, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- 0 – Message received.
- -ENOMSG – Returned without waiting.
- -EAGAIN – Waiting period timed out.

void k_mbox_data_get(struct *k_mbox_msg* *rx_msg, void *buffer)

Retrieve mailbox message data into a buffer.

This routine completes the processing of a received message by retrieving its data into a buffer, then disposing of the message.

Alternatively, this routine can be used to dispose of a received message without retrieving its data.

Parameters

- rx_msg – Address of the receive message descriptor.
- buffer – Address of the buffer to receive data, or NULL to discard the data.

Returns N/A

struct k_mbox_msg

#include <kernel.h> Mailbox Message Structure.

Public Members

size_t size

size of message (in bytes)

uint32_t info

application-defined information value

void *tx_data

sender's message data buffer

struct k_mem_block tx_block

message data block descriptor

k_tid_t rx_source_thread

source thread id

k_tid_t tx_target_thread

target thread id

struct k_mbox

#include <kernel.h> Mailbox Structure.

Public Members

_wait_q_t tx_msg_queue

Transmit messages queue

_wait_q_t rx_msg_queue

Receive message queue

Pipes

A *pipe* is a kernel object that allows a thread to send a byte stream to another thread. Pipes can be used to transfer chunks of data in whole or in part, and either synchronously or asynchronously.

Concepts The pipe can be configured with a ring buffer which holds data that has been sent but not yet received; alternatively, the pipe may have no ring buffer.

Any number of pipes can be defined (limited only by available RAM). Each pipe is referenced by its memory address.

A pipe has the following key property:

- A **size** that indicates the size of the pipe's ring buffer. Note that a size of zero defines a pipe with no ring buffer.

A pipe must be initialized before it can be used. The pipe is initially empty.

Data can be synchronously **sent** either in whole or in part to a pipe by a thread. If the specified minimum number of bytes can not be immediately satisfied, then the operation will either fail immediately or attempt to send as many bytes as possible and then pend in the hope that the send can be completed later. Accepted data is either copied to the pipe's ring buffer or directly to the waiting reader(s).

Data can be asynchronously **sent** in whole using a memory block to a pipe by a thread. Once the pipe has accepted all the bytes in the memory block, it will free the memory block and may give a semaphore if one was specified.

Data can be synchronously **received** from a pipe by a thread. If the specified minimum number of bytes can not be immediately satisfied, then the operation will either fail immediately or attempt to receive as many bytes as possible and then pend in the hope that the receive can be completed later. Accepted data is either copied from the pipe's ring buffer or directly from the waiting sender(s).

Note: The kernel does NOT allow for an ISR to send or receive data to/from a pipe even if it does not attempt to wait for space/data.

Implementation A pipe is defined using a variable of type *k_pipe* and an optional character buffer of type `unsigned char`. It must then be initialized by calling *k_pipe_init()*.

The following code defines and initializes an empty pipe that has a ring buffer capable of holding 100 bytes and is aligned to a 4-byte boundary.

```
unsigned char __aligned(4) my_ring_buffer[100];
struct k_pipe my_pipe;

k_pipe_init(&my_pipe, my_ring_buffer, sizeof(my_ring_buffer));
```

Alternatively, a pipe can be defined and initialized at compile time by calling *K_PIPE_DEFINE*.

The following code has the same effect as the code segment above. Observe that that macro defines both the pipe and its ring buffer.

```
K_PIPE_DEFINE(my_pipe, 100, 4);
```

Writing to a Pipe Data is added to a pipe by calling *k_pipe_put()*.

The following code builds on the example above, and uses the pipe to pass data from a producing thread to one or more consuming threads. If the pipe's ring buffer fills up because the consumers can't keep up, the producing thread waits for a specified amount of time.

```
struct message_header {
    ...
};

void producer_thread(void)
{
    unsigned char *data;
    size_t total_size;
    size_t bytes_written;
    int    rc;
    ...

    while (1) {
        /* Craft message to send in the pipe */
        data = ...;
        total_size = ...;

        /* send data to the consumers */
        rc = k_pipe_put(&my_pipe, data, total_size, &bytes_written,
                        sizeof(struct message_header), K_NO_WAIT);

        if (rc < 0) {
            /* Incomplete message header sent */
            ...
        } else if (bytes_written < total_size) {
            /* Some of the data was sent */
            ...
        } else {
            /* All data sent */
            ...
        }
    }
}
```

Reading from a Pipe Data is read from the pipe by calling *k_pipe_get()*.

The following code builds on the example above, and uses the pipe to process data items generated by one or more producing threads.

```
void consumer_thread(void)
{
    unsigned char buffer[120];
    size_t   bytes_read;
    struct message_header  *header = (struct message_header *)buffer;
```

(continues on next page)

(continued from previous page)

```
    while (1) {
        rc = k_pipe_get(&my_pipe, buffer, sizeof(buffer), &bytes_read,
                        sizeof(header), K_MSEC(100));

        if ((rc < 0) || (bytes_read < sizeof (header))) {
            /* Incomplete message header received */
            ...
        } else if (header->num_data_bytes + sizeof(header) > bytes_read) {
            /* Only some data was received */
            ...
        } else {
            /* All data was received */
            ...
        }
    }
}
```

Suggested uses Use a pipe to send streams of data between threads.

Note: A pipe can be used to transfer long streams of data if desired. However it is often preferable to send pointers to large data items to avoid copying the data.

Configuration Options Related configuration options:

- `CONFIG_NUM_PIPE_ASYNC_MSGS`

API Reference

group `pipe_apis`

Defines

`K_PIPE_DEFINE(name, pipe_buffer_size, pipe_align)`

Statically define and initialize a pipe.

The pipe can be accessed outside the module where it is defined using:

```
extern struct k_pipe <name>;
```

Parameters

- `name` – Name of the pipe.
- `pipe_buffer_size` – Size of the pipe's ring buffer (in bytes), or zero if no ring buffer is used.
- `pipe_align` – Alignment of the pipe's ring buffer (power of 2).

Functions

void k_pipe_init(struct *k_pipe* *pipe, unsigned char *buffer, size_t size)

Initialize a pipe.

This routine initializes a pipe object, prior to its first use.

Parameters

- pipe – Address of the pipe.
- buffer – Address of the pipe's ring buffer, or NULL if no ring buffer is used.
- size – Size of the pipe's ring buffer (in bytes), or zero if no ring buffer is used.

Returns N/A

int k_pipe_cleanup(struct *k_pipe* *pipe)

Release a pipe's allocated buffer.

If a pipe object was given a dynamically allocated buffer via *k_pipe_alloc_init()*, this will free it. This function does nothing if the buffer wasn't dynamically allocated.

Parameters

- pipe – Address of the pipe.

Return values

- 0 – on success
- -EAGAIN – nothing to cleanup

int k_pipe_alloc_init(struct *k_pipe* *pipe, size_t size)

Initialize a pipe and allocate a buffer for it.

Storage for the buffer region will be allocated from the calling thread's resource pool. This memory will be released if *k_pipe_cleanup()* is called, or userspace is enabled and the pipe object loses all references to it.

This function should only be called on uninitialized pipe objects.

Parameters

- pipe – Address of the pipe.
- size – Size of the pipe's ring buffer (in bytes), or zero if no ring buffer is used.

Return values

- 0 – on success
- -ENOMEM – if memory couldn't be allocated

int k_pipe_put(struct *k_pipe* *pipe, void *data, size_t bytes_to_write, size_t *bytes_written, size_t min_xfer, *k_timeout_t* timeout)

Write data to a pipe.

This routine writes up to *bytes_to_write* bytes of data to *pipe*.

Parameters

- pipe – Address of the pipe.
- data – Address of data to write.
- bytes_to_write – Size of data (in bytes).
- bytes_written – Address of area to hold the number of bytes written.
- min_xfer – Minimum number of bytes to write.
- timeout – Waiting period to wait for the data to be written, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – At least *min_xfer* bytes of data were written.
- `-EIO` – Returned without waiting; zero data bytes were written.
- `-EAGAIN` – Waiting period timed out; between zero and *min_xfer* minus one data bytes were written.

int `k_pipe_get`(struct *k_pipe* *pipe, void *data, size_t bytes_to_read, size_t *bytes_read, size_t min_xfer, *k_timeout_t* timeout)

Read data from a pipe.

This routine reads up to *bytes_to_read* bytes of data from *pipe*.

Parameters

- `pipe` – Address of the pipe.
- `data` – Address to place the data read from pipe.
- `bytes_to_read` – Maximum number of data bytes to read.
- `bytes_read` – Address of area to hold the number of bytes read.
- `min_xfer` – Minimum number of data bytes to read.
- `timeout` – Waiting period to wait for the data to be read, or one of the special values K_NO_WAIT and K_FOREVER.

Return values

- `0` – At least *min_xfer* bytes of data were read.
- `-EINVAL` – invalid parameters supplied
- `-EIO` – Returned without waiting; zero data bytes were read.
- `-EAGAIN` – Waiting period timed out; between zero and *min_xfer* minus one data bytes were read.

size_t `k_pipe_read_avail`(struct *k_pipe* *pipe)

Query the number of bytes that may be read from *pipe*.

Parameters

- `pipe` – Address of the pipe.

Return values `a` – number n such that 0 <= n <= *k_pipe::size*; the result is zero for unbuffered pipes.

size_t `k_pipe_write_avail`(struct *k_pipe* *pipe)

Query the number of bytes that may be written to *pipe*.

Parameters

- `pipe` – Address of the pipe.

Return values `a` – number n such that 0 <= n <= *k_pipe::size*; the result is zero for unbuffered pipes.

struct `k_pipe`

#include <kernel.h> Pipe Structure

Public Members

unsigned char *buffer
: Pipe buffer: may be NULL

size_t size
: Buffer size

size_t bytes_used

size_t read_index
: Where in buffer to read from

size_t write_index
: Where in buffer to write

struct *k_spinlock* lock
: Synchronization lock

_wait_q_t readers
: Reader wait queue

_wait_q_t writers
: Writer wait queue

uint8_t flags
: Wait queue Flags

7.13.3 Memory Management

These pages cover memory allocation and management services.

Memory Heaps

Zephyr provides a collection of utilities that allow threads to dynamically allocate memory.

Synchronized Heap Allocator

Creating a Heap The simplest way to define a heap is statically, with the *K_HEAP_DEFINE* macro. This creates a static *k_heap* variable with a given name that manages a memory region of the specified size.

Heaps can also be created to manage arbitrary regions of application-controlled memory using *k_heap_init()*.

Allocating Memory Memory can be allocated from a heap using *k_heap_alloc()*, passing it the address of the heap object and the number of bytes desired. This functions similarly to standard C `malloc()`, returning a NULL pointer on an allocation failure.

The heap supports blocking operation, allowing threads to go to sleep until memory is available. The final argument is a *k_timeout_t* timeout value indicating how long the thread may sleep before returning, or else one of the constant timeout values *K_NO_WAIT* or *K_FOREVER*.

Releasing Memory Memory allocated with *k_heap_alloc()* must be released using *k_heap_free()*. Similar to standard C `free()`, the pointer provided must be either a `NULL` value or a pointer previously returned by *k_heap_alloc()* for the same heap. Freeing a `NULL` value is defined to have no effect.

Low Level Heap Allocator The underlying implementation of the *k_heap* abstraction is provided a data structure named `sys_heap`. This implements exactly the same allocation semantics, but provides no kernel synchronization tools. It is available for applications that want to manage their own blocks of memory in contexts (for example, userspace) where synchronization is unavailable or more complicated. Unlike `k_heap`, all calls to any `sys_heap` functions on a single heap must be serialized by the caller. Simultaneous use from separate threads is disallowed.

Implementation Internally, the `sys_heap` memory block is partitioned into "chunks" of 8 bytes. All allocations are made out of a contiguous region of chunks. The first chunk of every allocation or unused block is prefixed by a chunk header that stores the length of the chunk, the length of the next lower ("left") chunk in physical memory, a bit indicating whether the chunk is in use, and chunk-indexed link pointers to the previous and next chunk in a "free list" to which unused chunks are added.

The heap code takes reasonable care to avoid fragmentation. Free block lists are stored in "buckets" by their size, each bucket storing blocks within one power of two (i.e. a bucket for blocks of 3-4 chunks, another for 5-8, 9-16, etc...) this allows new allocations to be made from the smallest/most-fragmented blocks available. Also, as allocations are freed and added to the heap, they are automatically combined with adjacent free blocks to prevent fragmentation.

All metadata is stored at the beginning of the contiguous block of heap memory, including the variable-length list of bucket list heads (which depend on heap size). The only external memory required is the `sys_heap` structure itself.

The `sys_heap` functions are unsynchronized. Care must be taken by any users to prevent concurrent access. Only one context may be inside one of the API functions at a time.

The heap code takes care to present high performance and reliable latency. All `sys_heap` API functions are guaranteed to complete within constant time. On typical architectures, they will all complete within 1-200 cycles. One complexity is that the search of the minimum bucket size for an allocation (the set of free blocks that "might fit") has a compile-time upper bound of iterations to prevent unbounded list searches, at the expense of some fragmentation resistance. This `CONFIG_SYS_HEAP_ALLOC_LOOPS` value may be chosen by the user at build time, and defaults to a value of 3.

System Heap The *system heap* is a predefined memory allocator that allows threads to dynamically allocate memory from a common memory region in a `malloc()`-like manner.

Only a single system heap is defined. Unlike other heaps or memory pools, the system heap cannot be directly referenced using its memory address.

The size of the system heap is configurable to arbitrary sizes, subject to space availability.

A thread can dynamically allocate a chunk of heap memory by calling *k_malloc()*. The address of the allocated chunk is guaranteed to be aligned on a multiple of pointer sizes. If a suitable chunk of heap memory cannot be found `NULL` is returned.

When the thread is finished with a chunk of heap memory it can release the chunk back to the system heap by calling *k_free()*.

Defining the Heap Memory Pool The size of the heap memory pool is specified using the `CONFIG_HEAP_MEM_POOL_SIZE` configuration option.

By default, the heap memory pool size is zero bytes. This value instructs the kernel not to define the heap memory pool object. The maximum size is limited by the amount of available memory in the system. The project build will fail in the link stage if the size specified can not be supported.

Allocating Memory A chunk of heap memory is allocated by calling *k_malloc()*.

The following code allocates a 200 byte chunk of heap memory, then fills it with zeros. A warning is issued if a suitable chunk is not obtained.

```
char *mem_ptr;

mem_ptr = k_malloc(200);
if (mem_ptr != NULL)) {
    memset(mem_ptr, 0, 200);
    ...
} else {
    printf("Memory not allocated");
}
```

Releasing Memory A chunk of heap memory is released by calling *k_free()*.

The following code allocates a 75 byte chunk of memory, then releases it once it is no longer needed.

```
char *mem_ptr;

mem_ptr = k_malloc(75);
... /* use memory block */
k_free(mem_ptr);
```

Suggested Uses Use the heap memory pool to dynamically allocate memory in a `malloc()`-like manner.

Configuration Options Related configuration options:

- `CONFIG_HEAP_MEM_POOL_SIZE`

API Reference

group `heap_apis`

Defines

`K_HEAP_DEFINE(name, bytes)`

Define a static *k_heap*.

This macro defines and initializes a static memory region and *k_heap* of the requested size. After kernel start, &name can be used as if *k_heap_init()* had been called.

Note that this macro enforces a minimum size on the memory region to accommodate metadata requirements. Very small heaps will be padded to fit.

Parameters

- `name` – Symbol name for the struct *k_heap* object

- `bytes` – Size of memory region, in bytes

`K_HEAP_DEFINE_NOCACHE(name, bytes)`

Define a static *k_heap* in uncached memory.

This macro defines and initializes a static memory region and *k_heap* of the requested size in uncache memory. After kernel start, &name can be used as if *k_heap_init()* had been called.

Note that this macro enforces a minimum size on the memory region to accommodate metadata requirements. Very small heaps will be padded to fit.

Parameters

- `name` – Symbol name for the struct *k_heap* object
- `bytes` – Size of memory region, in bytes

Functions

void `k_heap_init`(struct *k_heap* *h, void *mem, size_t bytes)

Initialize a *k_heap*.

This constructs a synchronized *k_heap* object over a memory region specified by the user. Note that while any alignment and size can be passed as valid parameters, internal alignment restrictions inside the inner sys_heap mean that not all bytes may be usable as allocated memory.

Parameters

- `h` – Heap struct to initialize
- `mem` – Pointer to memory.
- `bytes` – Size of memory region, in bytes

void *`k_heap_aligned_alloc`(struct *k_heap* *h, size_t align, size_t bytes, *k_timeout_t* timeout)

Allocate aligned memory from a *k_heap*.

Behaves in all ways like *k_heap_alloc()*, except that the returned memory (if available) will have a starting address in memory which is a multiple of the specified power-of-two alignment value in bytes. The resulting memory can be returned to the heap using *k_heap_free()*.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Note: When CONFIG_MULTITHREADING=n any *timeout* is treated as K_NO_WAIT.

Parameters

- `h` – Heap from which to allocate
- `align` – Alignment in bytes, must be a power of two
- `bytes` – Number of bytes requested
- `timeout` – How long to wait, or K_NO_WAIT

Returns Pointer to memory the caller can now use

void *k_heap_alloc(struct *k_heap* *h, size_t bytes, *k_timeout_t* timeout)

Allocate memory from a *k_heap*.

Allocates and returns a memory buffer from the memory region owned by the heap. If no memory is available immediately, the call will block for the specified timeout (constructed via the standard timeout API, or K_NO_WAIT or K_FOREVER) waiting for memory to be freed. If the allocation cannot be performed by the expiration of the timeout, NULL will be returned.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Note: When CONFIG_MULTITHREADING=n any *timeout* is treated as K_NO_WAIT.

Parameters

- `h` – Heap from which to allocate
- `bytes` – Desired size of block to allocate
- `timeout` – How long to wait, or K_NO_WAIT

Returns A pointer to valid heap memory, or NULL

void k_heap_free(struct *k_heap* *h, void *mem)

Free memory allocated by *k_heap_alloc()*

Returns the specified memory block, which must have been returned from *k_heap_alloc()*, to the heap for use by other callers. Passing a NULL block is legal, and has no effect.

Parameters

- `h` – Heap to which to return the memory
- `mem` – A valid memory block, or NULL

void *k_aligned_alloc(size_t align, size_t size)

Allocate memory from the heap with a specified alignment.

This routine provides semantics similar to aligned_alloc(); memory is allocated from the heap with a specified alignment. However, one minor difference is that *k_aligned_alloc()* accepts any non-zero `size`, wherase aligned_alloc() only accepts a `size` that is an integral multiple of `align`.

Above, aligned_alloc() refers to: C11 standard (ISO/IEC 9899:2011): 7.22.3.1 The aligned_alloc function (p: 347-348)

Parameters

- `align` – Alignment of memory requested (in bytes).
- `size` – Amount of memory requested (in bytes).

Returns Address of the allocated memory if successful; otherwise NULL.

void *k_malloc(size_t size)

Allocate memory from the heap.

This routine provides traditional malloc() semantics. Memory is allocated from the heap memory pool.

Parameters

- `size` – Amount of memory requested (in bytes).

Returns Address of the allocated memory if successful; otherwise NULL.

void `k_free`(void *ptr)

Free memory allocated from heap.

This routine provides traditional free() semantics. The memory being returned must have been allocated from the heap memory pool or k_mem_pool_malloc().

If *ptr* is NULL, no operation is performed.

Parameters

- `ptr` – Pointer to previously allocated memory.

Returns N/A

void *`k_calloc`(size_t nmemb, size_t size)

Allocate memory from heap, array style.

This routine provides traditional calloc() semantics. Memory is allocated from the heap memory pool and zeroed.

Parameters

- `nmemb` – Number of elements in the requested array
- `size` – Size of each array element (in bytes).

Returns Address of the allocated memory if successful; otherwise NULL.

struct `k_heap`

#include <kernel.h>

Memory Slabs

A *memory slab* is a kernel object that allows memory blocks to be dynamically allocated from a designated memory region. All memory blocks in a memory slab have a single fixed size, allowing them to be allocated and released efficiently and avoiding memory fragmentation concerns.

Concepts Any number of memory slabs can be defined (limited only by available RAM). Each memory slab is referenced by its memory address.

A memory slab has the following key properties:

- The **block size** of each block, measured in bytes. It must be at least 4N bytes long, where N is greater than 0.

- The **number of blocks** available for allocation. It must be greater than zero.
- A **buffer** that provides the memory for the memory slab's blocks. It must be at least "block size" times "number of blocks" bytes long.

The memory slab's buffer must be aligned to an N-byte boundary, where N is a power of 2 larger than 2 (i.e. 4, 8, 16, ...). To ensure that all memory blocks in the buffer are similarly aligned to this boundary, the block size must also be a multiple of N.

A memory slab must be initialized before it can be used. This marks all of its blocks as unused.

A thread that needs to use a memory block simply allocates it from a memory slab. When the thread finishes with a memory block, it must release the block back to the memory slab so the block can be reused.

If all the blocks are currently in use, a thread can optionally wait for one to become available. Any number of threads may wait on an empty memory slab simultaneously; when a memory block becomes available, it is given to the highest-priority thread that has waited the longest.

Unlike a heap, more than one memory slab can be defined, if needed. This allows for a memory slab with smaller blocks and others with larger-sized blocks. Alternatively, a memory pool object may be used.

Internal Operation A memory slab's buffer is an array of fixed-size blocks, with no wasted space between the blocks.

The memory slab keeps track of unallocated blocks using a linked list; the first 4 bytes of each unused block provide the necessary linkage.

Implementation

Defining a Memory Slab A memory slab is defined using a variable of type `k_mem_slab`. It must then be initialized by calling *k_mem_slab_init()*.

The following code defines and initializes a memory slab that has 6 blocks that are 400 bytes long, each of which is aligned to a 4-byte boundary..

```
struct k_mem_slab my_slab;
char __aligned(4) my_slab_buffer[6 * 400];

k_mem_slab_init(&my_slab, my_slab_buffer, 400, 6);
```

Alternatively, a memory slab can be defined and initialized at compile time by calling *K_MEM_SLAB_DEFINE*.

The following code has the same effect as the code segment above. Observe that the macro defines both the memory slab and its buffer.

```
K_MEM_SLAB_DEFINE(my_slab, 400, 6, 4);
```

Allocating a Memory Block A memory block is allocated by calling *k_mem_slab_alloc()*.

The following code builds on the example above, and waits up to 100 milliseconds for a memory block to become available, then fills it with zeroes. A warning is printed if a suitable block is not obtained.

```
char *block_ptr;

if (k_mem_slab_alloc(&my_slab, &block_ptr, 100) == 0)) {
    memset(block_ptr, 0, 400);
    ...
```

(continues on next page)

(continued from previous page)

```
} else {
    printf("Memory allocation time-out");
}
```

Releasing a Memory Block A memory block is released by calling *k_mem_slab_free()*.

The following code builds on the example above, and allocates a memory block, then releases it once it is no longer needed.

```
char *block_ptr;

k_mem_slab_alloc(&my_slab, &block_ptr, K_FOREVER);
... /* use memory block pointed at by block_ptr */
k_mem_slab_free(&my_slab, &block_ptr);
```

Suggested Uses Use a memory slab to allocate and free memory in fixed-size blocks.

Use memory slab blocks when sending large amounts of data from one thread to another, to avoid unnecessary copying of the data.

Configuration Options Related configuration options:

- `CONFIG_MEM_SLAB_TRACE_MAX_UTILIZATION`

API Reference

group `mem_slab_apis`

Defines

`K_MEM_SLAB_DEFINE(name, slab_block_size, slab_num_blocks, slab_align)`

Statically define and initialize a memory slab.

The memory slab's buffer contains *slab_num_blocks* memory blocks that are *slab_block_size* bytes long. The buffer is aligned to a *slab_align* -byte boundary. To ensure that each memory block is similarly aligned to this boundary, *slab_block_size* must also be a multiple of *slab_align*.

The memory slab can be accessed outside the module where it is defined using:

```
extern struct k_mem_slab <name>;
```

Parameters

- `name` – Name of the memory slab.
- `slab_block_size` – Size of each memory block (in bytes).
- `slab_num_blocks` – Number memory blocks.
- `slab_align` – Alignment of the memory slab's buffer (power of 2).

Functions

int `k_mem_slab_init`(struct k_mem_slab *slab, void *buffer, size_t block_size, uint32_t num_blocks)

Initialize a memory slab.

Initializes a memory slab, prior to its first use.

The memory slab's buffer contains *slab_num_blocks* memory blocks that are *slab_block_size* bytes long. The buffer must be aligned to an N-byte boundary matching a word boundary, where N is a power of 2 (i.e. 4 on 32-bit systems, 8, 16, ...). To ensure that each memory block is similarly aligned to this boundary, *slab_block_size* must also be a multiple of N.

Parameters

- `slab` – Address of the memory slab.
- `buffer` – Pointer to buffer used for the memory blocks.
- `block_size` – Size of each memory block (in bytes).
- `num_blocks` – Number of memory blocks.

Return values

- `0` – on success
- `-EINVAL` – invalid data supplied

int `k_mem_slab_alloc`(struct k_mem_slab *slab, void **mem, *k_timeout_t* timeout)

Allocate memory from a memory slab.

This routine allocates a memory block from a memory slab.

Function properties (list may not be complete) *isr-ok*

Note: *timeout* must be set to K_NO_WAIT if called from ISR.

Note: When CONFIG_MULTITHREADING=n any *timeout* is treated as K_NO_WAIT.

Parameters

- `slab` – Address of the memory slab.
- `mem` – Pointer to block address area.
- `timeout` – Non-negative waiting period to wait for operation to complete. Use K_NO_WAIT to return without waiting, or K_FOREVER to wait as long as necessary.

Return values

- `0` – Memory allocated. The block address area pointed at by *mem* is set to the starting address of the memory block.
- `-ENOMEM` – Returned without waiting.
- `-EAGAIN` – Waiting period timed out.
- `-EINVAL` – Invalid data supplied

void `k_mem_slab_free`(struct k_mem_slab *slab, void **mem)

Free memory allocated from a memory slab.

This routine releases a previously allocated memory block back to its associated memory slab.

Parameters

- `slab` – Address of the memory slab.
- `mem` – Pointer to block address area (as set by *k_mem_slab_alloc()*).

Returns N/A

static inline uint32_t `k_mem_slab_num_used_get`(struct k_mem_slab *slab)

Get the number of used blocks in a memory slab.

This routine gets the number of memory blocks that are currently allocated in *slab*.

Parameters

- `slab` – Address of the memory slab.

Returns Number of allocated memory blocks.

static inline uint32_t `k_mem_slab_max_used_get`(struct k_mem_slab *slab)

Get the number of maximum used blocks so far in a memory slab.

This routine gets the maximum number of memory blocks that were allocated in *slab*.

Parameters

- `slab` – Address of the memory slab.

Returns Maximum number of allocated memory blocks.

static inline uint32_t `k_mem_slab_num_free_get`(struct k_mem_slab *slab)

Get the number of unused blocks in a memory slab.

This routine gets the number of memory blocks that are currently unallocated in *slab*.

Parameters

- `slab` – Address of the memory slab.

Returns Number of unallocated memory blocks.

7.13.4 Timing

These pages cover timing related services.

Kernel Timing

Zephyr provides a robust and scalable timing framework to enable reporting and tracking of timed events from hardware timing sources of arbitrary precision.

Time Units Kernel time is tracked in several units which are used for different purposes.

Real time values, typically specified in milliseconds or microseconds, are the default presentation of time to application code. They have the advantages of being universally portable and pervasively understood, though they may not match the precision of the underlying hardware perfectly.

The kernel presents a "cycle" count via the *k_cycle_get_32()* API. The intent is that this counter represents the fastest cycle counter that the operating system is able to present to the user (for example, a CPU cycle counter) and that the read operation is very fast. The expectation is that very sensitive application

code might use this in a polling manner to achieve maximal precision. The frequency of this counter is required to be steady over time, and is available from `sys_clock_hw_cycles_per_sec()` (which on almost all platforms is a runtime constant that evaluates to CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC).

For asynchronous timekeeping, the kernel defines a "ticks" concept. A "tick" is the internal count in which the kernel does all its internal uptime and timeout bookkeeping. Interrupts are expected to be delivered on tick boundaries to the extent practical, and no fractional ticks are tracked. The choice of tick rate is configurable via `CONFIG_SYS_CLOCK_TICKS_PER_SEC`. Defaults on most hardware platforms (ones that support setting arbitrary interrupt timeouts) are expected to be in the range of 10 kHz, with software emulation platforms and legacy drivers using a more traditional 100 Hz value.

Conversion Zephyr provides an extensively enumerated conversion library with rounding control for all time units. Any unit of "ms" (milliseconds), "us" (microseconds), "tick", or "cyc" can be converted to any other. Control of rounding is provided, and each conversion is available in "floor" (round down to nearest output unit), "ceil" (round up) and "near" (round to nearest). Finally the output precision can be specified as either 32 or 64 bits.

For example: `k_ms_to_ticks_ceil32()` will convert a millisecond input value to the next higher number of ticks, returning a result truncated to 32 bits of precision; and `k_cyc_to_us_floor64()` will convert a measured cycle count to an elapsed number of microseconds in a full 64 bits of precision. See the reference documentation for the full enumeration of conversion routines.

On most platforms, where the various counter rates are integral multiples of each other and where the output fits within a single word, these conversions expand to a 2-4 operation sequence, requiring full precision only where actually required and requested.

Uptime The kernel tracks a system uptime count on behalf of the application. This is available at all times via *k_uptime_get()*, which provides an uptime value in milliseconds since system boot. This is expected to be the utility used by most portable application code.

The internal tracking, however, is as a 64 bit integer count of ticks. Apps with precise timing requirements (that are willing to do their own conversions to portable real time units) may access this with *k_uptime_ticks()*.

Timeouts The Zephyr kernel provides many APIs with a "timeout" parameter. Conceptually, this indicates the time at which an event will occur. For example:

- Kernel blocking operations like *k_sem_take()* or *k_queue_get()* may provide a timeout after which the routine will return with an error code if no data is available.
- Kernel `k_timer` objects must specify delays for their duration and period.
- The kernel *k_work_delayable* API provides a timeout parameter indicating when a work queue item will be added to the system queue.

All these values are specified using a *k_timeout_t* value. This is an opaque struct type that must be initialized using one of a family of kernel timeout macros. The most common, *K_MSEC* , defines a time in milliseconds after the current time (strictly: the time at which the kernel receives the timeout value).

Other options for timeout initialization follow the unit conventions described above: *K_NSEC()*, *K_USEC*, *K_TICKS* and *K_CYC()* specify timeout values that will expire after specified numbers of nanoseconds, microseconds, ticks and cycles, respectively.

Precision of *k_timeout_t* values is configurable, with the default being 32 bits. Large uptime counts in non-tick units will experience complicated rollover semantics, so it is expected that timing-sensitive applications with long uptimes will be configured to use a 64 bit timeout type.

Finally, it is possible to specify timeouts as absolute times since system boot. A timeout initialized with `K_TIMEOUT_ABS_MS` indicates a timeout that will expire after the system uptime reaches the specified value. There are likewise nanosecond, microsecond, cycles and ticks variants of this API.

Timing Internals

Timeout Queue All Zephyr *k_timeout_t* events specified using the API above are managed in a single, global queue of events. Each event is stored in a double-linked list, with an attendant delta count in ticks from the previous event. The action to take on an event is specified as a callback function pointer provided by the subsystem requesting the event, along with a `_timeout` tracking struct that is expected to be embedded within subsystem-defined data structures (for example: a `wait_q` struct, or a `k_tid_t` thread struct).

Note that all variant units passed via a *k_timeout_t* are converted to ticks once on insertion into the list. There no multiple-conversion steps internal to the kernel, so precision is guaranteed at the tick level no matter how many events exist or how long a timeout might be.

Note that the list structure means that the CPU work involved in managing large numbers of timeouts is quadratic in the number of active timeouts. The API design of the timeout queue was intended to permit a more scalable backend data structure, but no such implementation exists currently.

Timer Drivers Kernel timing at the tick level is driven by a timer driver with a comparatively simple API.

- The driver is expected to be able to "announce" new ticks to the kernel via the *sys_clock_announce()* call, which passes an integer number of ticks that have elapsed since the last announce call (or system boot). These calls can occur at any time, but the driver is expected to attempt to ensure (to the extent practical given interrupt latency interactions) that they occur near tick boundaries (i.e. not "halfway through" a tick), and most importantly that they be correct over time and subject to minimal skew vs. other counters and real world time.
- The driver is expected to provide a *sys_clock_set_timeout()* call to the kernel which indicates how many ticks may elapse before the kernel must receive an announce call to trigger registered timeouts. It is legal to announce new ticks before that moment (though they must be correct) but delay after that will cause events to be missed. Note that the timeout value passed here is in a delta from current time, but that does not absolve the driver of the requirement to provide ticks at a steady rate over time. Naive implementations of this function are subject to bugs where the fractional tick gets "reset" incorrectly and causes clock skew.
- The driver is expected to provide a *sys_clock_elapsed()* call which provides a current indication of how many ticks have elapsed (as compared to a real world clock) since the last call to *sys_clock_announce()*, which the kernel needs to test newly arriving timeouts for expiration.

Note that a natural implementation of this API results in a "tickless" kernel, which receives and processes timer interrupts only for registered events, relying on programmable hardware counters to provide irregular interrupts. But a traditional, "ticked" or "dumb" counter driver can be trivially implemented also:

- The driver can receive interrupts at a regular rate corresponding to the OS tick rate, calling *sys_clock_announce()* with an argument of one each time.
- The driver can ignore calls to *sys_clock_set_timeout()*, as every tick will be announced regardless of timeout status.
- The driver can return zero for every call to *sys_clock_elapsed()* as no more than one tick can be detected as having elapsed (because otherwise an interrupt would have been received).

SMP Details In general, the timer API described above does not change when run in a multiprocessor context. The kernel will internally synchronize all access appropriately, and ensure that all critical sections are small and minimal. But some notes are important to detail:

- Zephyr is agnostic about which CPU services timer interrupts. It is not illegal (though probably undesirable in some circumstances) to have every timer interrupt handled on a single processor. Existing SMP architectures implement symmetric timer drivers.

- The *sys_clock_announce()* call is expected to be globally synchronized at the driver level. The kernel does not do any per-CPU tracking, and expects that if two timer interrupts fire near simultaneously, that only one will provide the current tick count to the timing subsystem. The other may legally provide a tick count of zero if no ticks have elapsed. It should not "skip" the announce call because of timeslicing requirements (see below).
- Some SMP hardware uses a single, global timer device, others use a per-CPU counter. The complexity here (for example: ensuring counter synchronization between CPUs) is expected to be managed by the driver, not the kernel.
- The next timeout value passed back to the driver via *sys_clock_set_timeout()* is done identically for every CPU. So by default, every CPU will see simultaneous timer interrupts for every event, even though by definition only one of them should see a non-zero ticks argument to *sys_clock_announce()*. This is probably a correct default for timing sensitive applications (because it minimizes the chance that an errant ISR or interrupt lock will delay a timeout), but may be a performance problem in some cases. The current design expects that any such optimization is the responsibility of the timer driver.

Time Slicing An auxiliary job of the timing subsystem is to provide tick counters to the scheduler that allow implementation of time slicing of threads. A thread time-slice cannot be a timeout value, as it does not reflect a global expiration but instead a per-CPU value that needs to be tracked independently on each CPU in an SMP context.

Because there may be no other hardware available to drive timeslicing, Zephyr multiplexes the existing timer driver. This means that the value passed to *sys_clock_set_timeout()* may be clamped to a smaller value than the current next timeout when a time sliced thread is currently scheduled.

Subsystems that keep millisecond APIs In general, code like this will port just like applications code will. Millisecond values from the user may be treated any way the subsystem likes, and then converted into kernel timeouts using *K_MSEC()* at the point where they are presented to the kernel.

Obviously this comes at the cost of not being able to use new features, like the higher precision timeout constructors or absolute timeouts. But for many subsystems with simple needs, this may be acceptable.

One complexity is *K_FOREVER*. Subsystems that might have been able to accept this value to their millisecond API in the past no longer can, because it is no longer an intergral type. Such code will need to use a different, integer-valued token to represent "forever". *K_NO_WAIT* has the same typesafety concern too, of course, but as it is (and has always been) simply a numerical zero, it has a natural porting path.

Subsystems using `k_timeout_t` Ideally, code that takes a "timeout" parameter specifying a time to wait should be using the kernel native abstraction where possible. But *k_timeout_t* is opaque, and needs to be converted before it can be inspected by an application.

Some conversions are simple. Code that needs to test for *K_FOREVER* can simply use the *K_TIMEOUT_EQ()* macro to test the opaque struct for equality and take special action.

The more complicated case is when the subsystem needs to take a timeout and loop, waiting for it to finish while doing some processing that may require multiple blocking operations on underlying kernel code. For example, consider this design:

```
void my_wait_for_event(struct my_subsys *obj, int32_t timeout_in_ms)
{
    while (true) {
        uint32_t start = k_uptime_get_32();

        if (is_event_complete(obj)) {
            return;
        }
```

(continues on next page)

(continued from previous page)

```
        /* Wait for notification of state change */
        k_sem_take(obj->sem, timeout_in_ms);

        /* Subtract elapsed time */
        timeout_in_ms -= (k_uptime_get_32() - start);
    }
}
```

This code requires that the timeout value be inspected, which is no longer possible. For situations like this, the new API provides an internal `sys_clock_timeout_end_calc()` routine that converts an arbitrary timeout to the uptime value in ticks at which it will expire. So such a loop might look like:

```
void my_wait_for_event(struct my_subsys *obj, k_timeout_t timeout_in_ms)
{
    /* Compute the end time from the timeout */
    uint64_t end = sys_clock_timeout_end_calc(timeout_in_ms);

    while (end > k_uptime_ticks()) {
        if (is_event_complete(obj)) {
            return;
        }

        /* Wait for notification of state change */
        k_sem_take(obj->sem, timeout_in_ms);
    }
}
```

Note that `sys_clock_timeout_end_calc()` returns values in units of ticks, to prevent conversion aliasing, is always presented at 64 bit uptime precision to prevent rollover bugs, handles special *K_FOREVER* naturally (as `UINT64_MAX`), and works identically for absolute timeouts as well as conventional ones.

But some care is still required for subsystems that use it. Note that delta timeouts need to be interpreted relative to a "current time", and obviously that time is the time of the call to `sys_clock_timeout_end_calc()`. But the user expects that the time is the time they passed the timeout to you. Care must be taken to call this function just once, as synchronously as possible to the timeout creation in user code. It should not be used on a "stored" timeout value, and should never be called iteratively in a loop.

API Reference

group `clock_apis`

Clock APIs.

Defines

`K_NO_WAIT`

Generate null timeout delay.

This macro generates a timeout delay that instructs a kernel API not to wait if the requested operation cannot be performed immediately.

Returns Timeout delay value.

`K_NSEC(t)`

Generate timeout delay from nanoseconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* nanoseconds to perform the requested operation. Note that timer precision is limited to the tick rate, not the requested value.

Parameters

- `t` – Duration in nanoseconds.

Returns Timeout delay value.

`K_USEC(t)`

Generate timeout delay from microseconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* microseconds to perform the requested operation. Note that timer precision is limited to the tick rate, not the requested value.

Parameters

- `t` – Duration in microseconds.

Returns Timeout delay value.

`K_CYC(t)`

Generate timeout delay from cycles.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* cycles to perform the requested operation.

Parameters

- `t` – Duration in cycles.

Returns Timeout delay value.

`K_TICKS(t)`

Generate timeout delay from system ticks.

This macro generates a timeout delay that instructs a kernel API to wait up to *t* ticks to perform the requested operation.

Parameters

- `t` – Duration in system ticks.

Returns Timeout delay value.

`K_MSEC(ms)`

Generate timeout delay from milliseconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *ms* milliseconds to perform the requested operation.

Parameters

- `ms` – Duration in milliseconds.

Returns Timeout delay value.

`K_SECONDS(s)`

Generate timeout delay from seconds.

This macro generates a timeout delay that instructs a kernel API to wait up to *s* seconds to perform the requested operation.

Parameters

- `s` – Duration in seconds.

Returns Timeout delay value.

K_MINUTES(m)

Generate timeout delay from minutes.

This macro generates a timeout delay that instructs a kernel API to wait up to *m* minutes to perform the requested operation.

Parameters

- m – Duration in minutes.

Returns Timeout delay value.

K_HOURS(h)

Generate timeout delay from hours.

This macro generates a timeout delay that instructs a kernel API to wait up to *h* hours to perform the requested operation.

Parameters

- h – Duration in hours.

Returns Timeout delay value.

K_FOREVER

Generate infinite timeout delay.

This macro generates a timeout delay that instructs a kernel API to wait as long as necessary to perform the requested operation.

Returns Timeout delay value.

K_TICKS_FOREVER

K_TIMEOUT_EQ(a, b)

Compare timeouts for equality.

The *k_timeout_t* object is an opaque struct that should not be inspected by application code. This macro exists so that users can test timeout objects for equality with known constants (e.g. K_NO_WAIT and K_FOREVER) when implementing their own APIs in terms of Zephyr timeout constants.

Returns True if the timeout objects are identical

Typedefs

typedef uint32_t k_ticks_t

Tick precision used in timeout APIs.

This type defines the word size of the timeout values used in *k_timeout_t* objects, and thus defines an upper bound on maximum timeout length (or equivalently minimum tick duration). Note that this does not affect the size of the system uptime counter, which is always a 64 bit count of ticks.

Functions

int sys_clock_driver_init(const struct *device* *dev)

Initialize system clock driver.

The system clock is a Zephyr device created globally. This is its initialization callback. It is a weak symbol that will be implemented as a noop if undefined in the clock driver.

int `clock_device_ctrl`(const struct *device* *dev, enum *pm_device_state* state)

Initialize system clock driver.

The system clock is a Zephyr device created globally. This is its device control callback, used in a few devices for power management. It is a weak symbol that will be implemented as a noop if undefined in the clock driver.

void `sys_clock_set_timeout`(int32_t ticks, bool idle)

Set system clock timeout.

Informs the system clock driver that the next needed call to *sys_clock_announce()* will not be until the specified number of ticks from the the current time have elapsed. Note that spurious calls to *sys_clock_announce()* are allowed (i.e. it's legal to announce every tick and implement this function as a noop), the requirement is that one tick announcement should occur within one tick BEFORE the specified expiration (that is, passing ticks==1 means "announce

the next tick", this convention was chosen to match legacy usage). Similarly a ticks value of zero (or even negative) is legal and treated identically: it simply indicates the kernel would like the next tick announcement as soon as possible.

Note that ticks can also be passed the special value K_TICKS_FOREVER, indicating that no future timer interrupts are expected or required and that the system is permitted to enter an indefinite sleep even if this could cause rollover of the internal counter (i.e. the system uptime counter is allowed to be wrong

Note also that it is conventional for the kernel to pass INT_MAX for ticks if it wants to preserve the uptime tick count but doesn't have a specific event to await. The intent here is that the driver will schedule any needed timeout as far into the future as possible. For the specific case of INT_MAX, the next call to *sys_clock_announce()* may occur at any point in the future, not just at INT_MAX ticks. But the correspondence between the announced ticks and real-world time must be correct.

A final note about SMP: note that the call to *sys_clock_set_timeout()* is made on any CPU, and reflects the next timeout desired globally. The resulting calls(s) to *sys_clock_announce()* must be properly serialized by the driver such that a given tick is announced exactly once across the system. The kernel does not (cannot, really) attempt to serialize things by "assigning" timeouts to specific CPUs.

Parameters

- `ticks` – Timeout in tick units
- `idle` – Hint to the driver that the system is about to enter the idle state immediately after setting the timeout

void `sys_clock_idle_exit`(void)

Timer idle exit notification.

This notifies the timer driver that the system is exiting the idle and allows it to do whatever bookkeeping is needed to restore timer operation and compute elapsed ticks.

Note: Legacy timer drivers also use this opportunity to call back into *sys_clock_announce()* to notify the kernel of expired ticks. This is allowed for compatibility, but not recommended. The kernel will figure that out on its own.

---

void `sys_clock_announce`(int32_t ticks)

Announce time progress to the kernel.

Informs the kernel that the specified number of ticks have elapsed since the last call to *sys_clock_announce()* (or system startup for the first call). The timer driver is expected to delivery these announcements as close as practical (subject to hardware and latency limitations) to tick boundaries.

Parameters

- `ticks` – Elapsed time, in ticks

uint32_t `sys_clock_elapsed`(void)

Ticks elapsed since last *sys_clock_announce()* call.

Queries the clock driver for the current time elapsed since the last call to *sys_clock_announce()* was made. The kernel will call this with appropriate locking, the driver needs only provide an instantaneous answer.

int64_t `k_uptime_ticks`(void)

Get system uptime, in system ticks.

This routine returns the elapsed time since the system booted, in ticks (c.f. `CONFIG_SYS_CLOCK_TICKS_PER_SEC`), which is the fundamental unit of resolution of kernel timekeeping.

Returns Current uptime in ticks.

static inline int64_t `k_uptime_get`(void)

Get system uptime.

This routine returns the elapsed time since the system booted, in milliseconds.

Note: While this function returns time in milliseconds, it does not mean it has millisecond resolution. The actual resolution depends on `CONFIG_SYS_CLOCK_TICKS_PER_SEC` config option.

Returns Current uptime in milliseconds.

static inline uint32_t `k_uptime_get_32`(void)

Get system uptime (32-bit version).

This routine returns the lower 32 bits of the system uptime in milliseconds.

Because correct conversion requires full precision of the system clock there is no benefit to using this over *k_uptime_get()* unless you know the application will never run long enough for the system clock to approach 2^32 ticks. Calls to this function may involve interrupt blocking and 64-bit math.

Note: While this function returns time in milliseconds, it does not mean it has millisecond resolution. The actual resolution depends on `CONFIG_SYS_CLOCK_TICKS_PER_SEC` config option

---

Returns The low 32 bits of the current uptime, in milliseconds.

static inline int64_t `k_uptime_delta`(int64_t *reftime)

Get elapsed time.

This routine computes the elapsed time between the current system uptime and an earlier reference time, in milliseconds.

Parameters

- `reftime` – Pointer to a reference time, which is updated to the current uptime upon return.

Returns Elapsed time.

static inline uint32_t k_cycle_get_32(void)

Read the hardware clock.

This routine returns the current time, as measured by the system's hardware clock.

Returns Current hardware clock up-counter (in cycles).

struct k_timeout_t

#include <sys_clock.h> Kernel timeout type.

Timeout arguments presented to kernel APIs are stored in this opaque type, which is capable of representing times in various formats and units. It should be constructed from application data using one of the macros defined for this purpose (e.g. *K_MSEC()*, K_TIMEOUT_ABS_TICKS(), etc...), or be one of the two constants K_NO_WAIT or K_FOREVER. Applications should not inspect the internal data once constructed. Timeout values may be compared for equality with the *K_TIMEOUT_EQ()* macro.

Timers

A *timer* is a kernel object that measures the passage of time using the kernel's system clock. When a timer's specified time limit is reached it can perform an application-defined action, or it can simply record the expiration and wait for the application to read its status.

Concepts Any number of timers can be defined (limited only by available RAM). Each timer is referenced by its memory address.

A timer has the following key properties:

- A *duration* specifying the time interval before the timer expires for the first time. This is a k_timeout_t value that may be initialized via different units.
- A *period* specifying the time interval between all timer expirations after the first one, also a k_timeout_t. It must be non-negative. A period of K_NO_WAIT (i.e. zero) or K_FOREVER means that the timer is a one shot timer that stops after a single expiration. (For example then, if a timer is started with a duration of 200 and a period of 75, it will first expire after 200ms and then every 75ms after that.)
- An *expiry function* that is executed each time the timer expires. The function is executed by the system clock interrupt handler. If no expiry function is required a NULL function can be specified.
- A *stop function* that is executed if the timer is stopped prematurely while running. The function is executed by the thread that stops the timer. If no stop function is required a NULL function can be specified.

- A *status* value that indicates how many times the timer has expired since the status value was last read.

A timer must be initialized before it can be used. This specifies its expiry function and stop function values, sets the timer's status to zero, and puts the timer into the **stopped** state.

A timer is **started** by specifying a duration and a period. The timer's status is reset to zero, then the timer enters the **running** state and begins counting down towards expiry.

Note that the timer's duration and period parameters specify **minimum** delays that will elapse. Because of internal system timer precision (and potentially runtime interactions like interrupt delay) it is possible that more time may have passed as measured by reads from the relevant system time APIs. But at least this much time is guaranteed to have elapsed.

When a running timer expires its status is incremented and the timer executes its expiry function, if one exists; If a thread is waiting on the timer, it is unblocked. If the timer's period is zero the timer enters the stopped state; otherwise the timer restarts with a new duration equal to its period.

A running timer can be stopped in mid-countdown, if desired. The timer's status is left unchanged, then the timer enters the stopped state and executes its stop function, if one exists. If a thread is waiting on the timer, it is unblocked. Attempting to stop a non-running timer is permitted, but has no effect on the timer since it is already stopped.

A running timer can be restarted in mid-countdown, if desired. The timer's status is reset to zero, then the timer begins counting down using the new duration and period values specified by the caller. If a thread is waiting on the timer, it continues waiting.

A timer's status can be read directly at any time to determine how many times the timer has expired since its status was last read. Reading a timer's status resets its value to zero. The amount of time remaining before the timer expires can also be read; a value of zero indicates that the timer is stopped.

A thread may read a timer's status indirectly by **synchronizing** with the timer. This blocks the thread until the timer's status is non-zero (indicating that it has expired at least once) or the timer is stopped; if the timer status is already non-zero or the timer is already stopped the thread continues without waiting. The synchronization operation returns the timer's status and resets it to zero.

Note: Only a single user should examine the status of any given timer, since reading the status (directly or indirectly) changes its value. Similarly, only a single thread at a time should synchronize with a given timer. ISRs are not permitted to synchronize with timers, since ISRs are not allowed to block.

Implementation

Defining a Timer A timer is defined using a variable of type `k_timer`. It must then be initialized by calling *`k_timer_init()`*.

The following code defines and initializes a timer.

```
struct k_timer my_timer;
extern void my_expiry_function(struct k_timer *timer_id);

k_timer_init(&my_timer, my_expiry_function, NULL);
```

Alternatively, a timer can be defined and initialized at compile time by calling *`K_TIMER_DEFINE`*.

The following code has the same effect as the code segment above.

```
K_TIMER_DEFINE(my_timer, my_expiry_function, NULL);
```

Using a Timer Expiry Function The following code uses a timer to perform a non-trivial action on a periodic basis. Since the required work cannot be done at interrupt level, the timer's expiry function submits a work item to the *system workqueue*, whose thread performs the work.

```
void my_work_handler(struct k_work *work)
{
    /* do the processing that needs to be done periodically */
    ...
}

K_WORK_DEFINE(my_work, my_work_handler);

void my_timer_handler(struct k_timer *dummy)
{
    k_work_submit(&my_work);
}

K_TIMER_DEFINE(my_timer, my_timer_handler, NULL);

...

/* start periodic timer that expires once every second */
k_timer_start(&my_timer, K_SECONDS(1), K_SECONDS(1));
```

Reading Timer Status The following code reads a timer's status directly to determine if the timer has expired on not.

```
K_TIMER_DEFINE(my_status_timer, NULL, NULL);

...

/* start one shot timer that expires after 200 ms */
k_timer_start(&my_status_timer, K_MSEC(200), K_NO_WAIT);

/* do work */
...

/* check timer status */
if (k_timer_status_get(&my_status_timer) > 0) {
    /* timer has expired */
} else if (k_timer_remaining_get(&my_status_timer) == 0) {
    /* timer was stopped (by someone else) before expiring */
} else {
    /* timer is still running */
}
```

Using Timer Status Synchronization The following code performs timer status synchronization to allow a thread to do useful work while ensuring that a pair of protocol operations are separated by the specified time interval.

```
K_TIMER_DEFINE(my_sync_timer, NULL, NULL);

...

/* do first protocol operation */
...
```

(continues on next page)

(continued from previous page)

```
/* start one shot timer that expires after 500 ms */
k_timer_start(&my_sync_timer, K_MSEC(500), K_NO_WAIT);

/* do other work */
...

/* ensure timer has expired (waiting for expiry, if necessary) */
k_timer_status_sync(&my_sync_timer);

/* do second protocol operation */
...
```

Note: If the thread had no other work to do it could simply sleep between the two protocol operations, without using a timer.

Suggested Uses Use a timer to initiate an asynchronous operation after a specified amount of time.

Use a timer to determine whether or not a specified amount of time has elapsed. In particular, timers should be used when higher precision and/or unit control is required than that afforded by the simpler *k_sleep()* and *k_usleep()* calls.

Use a timer to perform other work while carrying out operations involving time limits.

Note: If a thread needs to measure the time required to perform an operation it can read the *system clock or the hardware clock* directly, rather than using a timer.

Configuration Options Related configuration options:

- None

API Reference

group `timer_apis`

Defines

`K_TIMER_DEFINE(name, expiry_fn, stop_fn)`

Statically define and initialize a timer.

The timer can be accessed outside the module where it is defined using:

```
extern struct k_timer <name>;
```

Parameters

- `name` – Name of the timer variable.
- `expiry_fn` – Function to invoke each time the timer expires.
- `stop_fn` – Function to invoke if the timer is stopped while running.

Typedefs

typedef void (*`k_timer_expiry_t`)(struct k_timer *timer)

Timer expiry function type.

A timer's expiry function is executed by the system clock interrupt handler each time the timer expires. The expiry function is optional, and is only invoked if the timer has been initialized with one.

Param timer Address of timer.

Return N/A

typedef void (*`k_timer_stop_t`)(struct k_timer *timer)

Timer stop function type.

A timer's stop function is executed if the timer is stopped prematurely. The function runs in the context of call that stops the timer. As *k_timer_stop()* can be invoked from an ISR, the stop function must be callable from interrupt context (isr-ok).

The stop function is optional, and is only invoked if the timer has been initialized with one.

Param timer Address of timer.

Return N/A

Functions

void `k_timer_init`(struct k_timer *timer, *k_timer_expiry_t* expiry_fn, *k_timer_stop_t* stop_fn)

Initialize a timer.

This routine initializes a timer, prior to its first use.

Parameters

- `timer` – Address of timer.
- `expiry_fn` – Function to invoke each time the timer expires.
- `stop_fn` – Function to invoke if the timer is stopped while running.

Returns N/A

void `k_timer_start`(struct k_timer *timer, *k_timeout_t* duration, *k_timeout_t* period)

Start a timer.

This routine starts a timer, and resets its status to zero. The timer begins counting down using the specified duration and period values.

Attempting to start a timer that is already running is permitted. The timer's status is reset to zero and the timer begins counting down using the new duration and period values.

Parameters

- `timer` – Address of timer.
- `duration` – Initial timer duration.
- `period` – Timer period.

Returns N/A

void `k_timer_stop`(struct k_timer *timer)

Stop a timer.

This routine stops a running timer prematurely. The timer's stop function, if one exists, is invoked by the caller.

Attempting to stop a timer that is not running is permitted, but has no effect on the timer.

Function properties (list may not be complete) *isr-ok*

Note: The stop handler has to be callable from ISRs if *k_timer_stop* is to be called from ISRs.

Parameters

- `timer` – Address of timer.

Returns N/A

uint32_t `k_timer_status_get`(struct k_timer *timer)

Read timer status.

This routine reads the timer's status, which indicates the number of times it has expired since its status was last read.

Calling this routine resets the timer's status to zero.

Parameters

- `timer` – Address of timer.

Returns Timer status.

uint32_t `k_timer_status_sync`(struct k_timer *timer)

Synchronize thread to timer expiration.

This routine blocks the calling thread until the timer's status is non-zero (indicating that it has expired at least once since it was last examined) or the timer is stopped. If the timer status is already non-zero, or the timer is already stopped, the caller continues without waiting.

Calling this routine resets the timer's status to zero.

This routine must not be used by interrupt handlers, since they are not allowed to block.

Parameters

- `timer` – Address of timer.

Returns Timer status.

k_ticks_t `k_timer_expires_ticks`(const struct k_timer *timer)

Get next expiration time of a timer, in system ticks.

This routine returns the future system uptime reached at the next time of expiration of the timer, in units of system ticks. If the timer is not running, current system time is returned.

Parameters

- `timer` – The timer object

Returns Uptime of expiration, in ticks

k_ticks_t `k_timer_remaining_ticks`(const struct k_timer *timer)

Get time remaining before a timer next expires, in system ticks.

This routine computes the time remaining before a running timer next expires, in units of system ticks. If the timer is not running, it returns zero.

static inline uint32_t `k_timer_remaining_get`(struct k_timer *timer)

Get time remaining before a timer next expires.

This routine computes the (approximate) time remaining before a running timer next expires. If the timer is not running, it returns zero.

Parameters

- `timer` – Address of timer.

Returns Remaining time (in milliseconds).

void `k_timer_user_data_set`(struct k_timer *timer, void *user_data)

Associate user-specific data with a timer.

This routine records the *user_data* with the *timer*, to be retrieved later.

It can be used e.g. in a timer handler shared across multiple subsystems to retrieve data specific to the subsystem this timer is associated with.

Parameters

- `timer` – Address of timer.
- `user_data` – User data to associate with the timer.

Returns N/A

void *`k_timer_user_data_get`(const struct k_timer *timer)

Retrieve the user-specific data from a timer.

Parameters

- `timer` – Address of timer.

Returns The user data.

7.13.5 Other

These pages cover other kernel services.

CPU Idling

Although normally reserved for the idle thread, in certain special applications, a thread might want to make the CPU idle.

Concepts Making the CPU idle causes the kernel to pause all operations until an event, normally an interrupt, wakes up the CPU. In a regular system, the idle thread is responsible for this. However, in some constrained systems, it is possible that another thread takes this duty.

Implementation

Making the CPU idle Making the CPU idle is simple: call the k_cpu_idle() API. The CPU will stop executing instructions until an event occurs. Most likely, the function will be called within a loop. Note that in certain architectures, upon return, k_cpu_idle() unconditionally unmasks interrupts.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

void main(void)
{
    k_sem_init(&my_sem, 0, 1);

    /* wait for semaphore from ISR, then do related work */

    for (;;) {

        /* wait for ISR to trigger work to perform */
        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            /* ... do processing */

        }

        /* put CPU to sleep to save power */
        k_cpu_idle();
    }
}
```

Making the CPU idle in an atomic fashion It is possible that there is a need to do some work atomically before making the CPU idle. In such a case, k_cpu_atomic_idle() should be used instead.

In fact, there is a race condition in the previous example: the interrupt could occur between the time the semaphore is taken, finding out it is not available and making the CPU idle again. In some systems, this can cause the CPU to idle until *another* interrupt occurs, which might be *never*, thus hanging the system completely. To prevent this, k_cpu_atomic_idle() should have been used, like in this example.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

void main(void)
{
    k_sem_init(&my_sem, 0, 1);

    for (;;) {

        unsigned int key = irq_lock();

        /*
         * Wait for semaphore from ISR; if acquired, do related work, then
         * go to next loop iteration (the semaphore might have been given
```

(continues on next page)

(continued from previous page)

```
         * again); else, make the CPU idle.
         */

        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            irq_unlock(key);

            /* ... do processing */


        } else {
            /* put CPU to sleep to save power */
            k_cpu_atomic_idle(key);
        }
    }
}
```

Suggested Uses Use k_cpu_atomic_idle() when a thread has to do some real work in addition to idling the CPU to wait for an event. See example above.

Use k_cpu_idle() only when a thread is only responsible for idling the CPU, i.e. not doing any real work, like in this example below.

```
void main(void)
{
    /* ... do some system/application initialization */


    /* thread is only used for CPU idling from this point on */
    for (;;) {
        k_cpu_idle();
    }
}
```

Note: Do not use these APIs unless absolutely necessary. In a normal system, the idle thread takes care of power management, including CPU idling.

API Reference

group `cpu_idle_apis`

Functions

static inline void `k_cpu_idle(void)`

Make the CPU idle.

This function makes the CPU idle until an event wakes it up.

In a regular system, the idle thread should be the only thread responsible for making the CPU idle and triggering any type of power management. However, in some more constrained systems, such as a single-threaded system, the only thread would be responsible for this if needed.

Note: In some architectures, before returning, the function unmasks interrupts unconditionally.

Returns N/A

static inline void `k_cpu_atomic_idle`(unsigned int key)

Make the CPU idle in an atomic fashion.

Similar to *k_cpu_idle()*, but must be called with interrupts locked.

Enabling interrupts and entering a low-power mode will be atomic, i.e. there will be no period of time where interrupts are enabled before the processor enters a low-power mode.

After waking up from the low-power mode, the interrupt lockout state will be restored as if by *irq_unlock(key)*.

Parameters

- `key` – Interrupt locking key obtained from *irq_lock()*.

Returns N/A

Atomic Services

An *atomic variable* is a 32-bit variable that can be read and modified by threads and ISRs in an uninterruptible manner.

Concepts Any number of atomic variables can be defined (limited only by available RAM).

Using the kernel's atomic APIs to manipulate an atomic variable guarantees that the desired operation occurs correctly, even if higher priority contexts also manipulate the same variable.

The kernel also supports the atomic manipulation of a single bit in an array of atomic variables.

Implementation

Defining an Atomic Variable An atomic variable is defined using a variable of type `atomic_t`.

By default an atomic variable is initialized to zero. However, it can be given a different value using *ATOMIC_INIT*:

```
atomic_t flags = ATOMIC_INIT(0xFF);
```

Manipulating an Atomic Variable An atomic variable is manipulated using the APIs listed at the end of this section.

The following code shows how an atomic variable can be used to keep track of the number of times a function has been invoked. Since the count is incremented atomically, there is no risk that it will become corrupted in mid-increment if a thread calling the function is interrupted if by a higher priority context that also calls the routine.

```
atomic_t call_count;

int call_counting_routine(void)
{
    /* increment invocation counter */
    atomic_inc(&call_count);

    /* do rest of routine's processing */
    ...
}
```

Manipulating an Array of Atomic Variables An array of 32-bit atomic variables can be defined in the conventional manner. However, you can also define an N-bit array of atomic variables using *ATOMIC_DEFINE*.

A single bit in array of atomic variables can be manipulated using the APIs listed at the end of this section that end with `_bit()`.

The following code shows how a set of 200 flag bits can be implemented using an array of atomic variables.

```
#define NUM_FLAG_BITS 200

ATOMIC_DEFINE(flag_bits, NUM_FLAG_BITS);

/* set specified flag bit & return its previous value */
int set_flag_bit(int bit_position)
{
    return (int)atomic_set_bit(flag_bits, bit_position);
}
```

Suggested Uses Use an atomic variable to implement critical section processing that only requires the manipulation of a single 32-bit value.

Use multiple atomic variables to implement critical section processing on a set of flag bits in a bit array longer than 32 bits.

Note: Using atomic variables is typically far more efficient than using other techniques to implement critical sections such as using a mutex or locking interrupts.

Configuration Options Related configuration options:

- `CONFIG_ATOMIC_OPERATIONS_BUILTIN`
- `CONFIG_ATOMIC_OPERATIONS_ARCH`
- `CONFIG_ATOMIC_OPERATIONS_C`

API Reference

Important: All atomic services APIs can be used by both threads and ISRs.

group atomic_apis

Defines

ATOMIC_INIT(i)

Initialize an atomic variable.

This macro can be used to initialize an atomic variable. For example,

```
atomic_t my_var = ATOMIC_INIT(75);
```

Parameters

- i – Value to assign to atomic variable.

ATOMIC_PTR_INIT(p)

Initialize an atomic pointer variable.

This macro can be used to initialize an atomic pointer variable. For example,

```
atomic_ptr_t my_ptr = ATOMIC_PTR_INIT(&data);
```

Parameters

- p – Pointer value to assign to atomic pointer variable.

ATOMIC_BITMAP_SIZE(num_bits)

This macro computes the number of atomic variables necessary to represent a bitmap with *num_bits*.

Parameters

- num_bits – Number of bits.

ATOMIC_DEFINE(name, num_bits)

Define an array of atomic variables.

This macro defines an array of atomic variables containing at least *num_bits* bits.

Note: If used from file scope, the bits of the array are initialized to zero; if used from within a function, the bits are left uninitialized.

Parameters

- name – Name of array of atomic variables.
- num_bits – Number of bits needed.

Functions

static inline bool `atomic_test_bit`(const atomic_t *target, int bit)

Atomically test a bit.

This routine tests whether bit number *bit* of *target* is set or not. The target may be a single atomic variable or an array of them.

Parameters

- `target` – Address of atomic variable or array.
- `bit` – Bit number (starting from 0).

Returns true if the bit was set, false if it wasn't.

static inline bool `atomic_test_and_clear_bit`(atomic_t *target, int bit)

Atomically test and clear a bit.

Atomically clear bit number *bit* of *target* and return its old value. The target may be a single atomic variable or an array of them.

Parameters

- `target` – Address of atomic variable or array.
- `bit` – Bit number (starting from 0).

Returns true if the bit was set, false if it wasn't.

static inline bool `atomic_test_and_set_bit`(atomic_t *target, int bit)

Atomically set a bit.

Atomically set bit number *bit* of *target* and return its old value. The target may be a single atomic variable or an array of them.

Parameters

- `target` – Address of atomic variable or array.
- `bit` – Bit number (starting from 0).

Returns true if the bit was set, false if it wasn't.

static inline void `atomic_clear_bit`(atomic_t *target, int bit)

Atomically clear a bit.

Atomically clear bit number *bit* of *target*. The target may be a single atomic variable or an array of them.

Parameters

- `target` – Address of atomic variable or array.
- `bit` – Bit number (starting from 0).

Returns N/A

static inline void `atomic_set_bit`(atomic_t *target, int bit)

Atomically set a bit.

Atomically set bit number *bit* of *target*. The target may be a single atomic variable or an array of them.

Parameters

- `target` – Address of atomic variable or array.
- `bit` – Bit number (starting from 0).

Returns N/A

static inline void atomic_set_bit_to(atomic_t *target, int bit, bool val)

Atomically set a bit to a given value.

Atomically set bit number *bit* of *target* to value *val*. The target may be a single atomic variable or an array of them.

Parameters

- `target` – Address of atomic variable or array.
- `bit` – Bit number (starting from 0).
- `val` – true for 1, false for 0.

Returns N/A

static inline bool atomic_cas(atomic_t *target, atomic_val_t old_value, atomic_val_t new_value)

Atomic compare-and-set.

This routine performs an atomic compare-and-set on *target*. If the current value of *target* equals *old_value*, *target* is set to *new_value*. If the current value of *target* does not equal *old_value*, *target* is left unchanged.

Parameters

- `target` – Address of atomic variable.
- `old_value` – Original value to compare against.
- `new_value` – New value to store.

Returns true if *new_value* is written, false otherwise.

static inline bool atomic_ptr_cas(atomic_ptr_t *target, atomic_ptr_val_t old_value,
atomic_ptr_val_t new_value)

Atomic compare-and-set with pointer values.

This routine performs an atomic compare-and-set on *target*. If the current value of *target* equals *old_value*, *target* is set to *new_value*. If the current value of *target* does not equal *old_value*, *target* is left unchanged.

Parameters

- `target` – Address of atomic variable.
- `old_value` – Original value to compare against.
- `new_value` – New value to store.

Returns true if *new_value* is written, false otherwise.

static inline atomic_val_t atomic_add(atomic_t *target, atomic_val_t value)

Atomic addition.

This routine performs an atomic addition on *target*.

Parameters

- `target` – Address of atomic variable.
- `value` – Value to add.

Returns Previous value of *target*.

static inline atomic_val_t atomic_sub(atomic_t *target, atomic_val_t value)

Atomic subtraction.

This routine performs an atomic subtraction on *target*.

Parameters

- `target` – Address of atomic variable.

- `value` – Value to subtract.

Returns Previous value of *target*.

static inline atomic_val_t `atomic_inc`(atomic_t *target)

Atomic increment.

This routine performs an atomic increment by 1 on *target*.

Parameters

- `target` – Address of atomic variable.

Returns Previous value of *target*.

static inline atomic_val_t `atomic_dec`(atomic_t *target)

Atomic decrement.

This routine performs an atomic decrement by 1 on *target*.

Parameters

- `target` – Address of atomic variable.

Returns Previous value of *target*.

static inline atomic_val_t `atomic_get`(const atomic_t *target)

Atomic get.

This routine performs an atomic read on *target*.

Parameters

- `target` – Address of atomic variable.

Returns Value of *target*.

static inline atomic_ptr_val_t `atomic_ptr_get`(const atomic_ptr_t *target)

Atomic get a pointer value.

This routine performs an atomic read on *target*.

Parameters

- `target` – Address of pointer variable.

Returns Value of *target*.

static inline atomic_val_t `atomic_set`(atomic_t *target, atomic_val_t value)

Atomic get-and-set.

This routine atomically sets *target* to *value* and returns the previous value of *target*.

Parameters

- `target` – Address of atomic variable.
- `value` – Value to write to *target*.

Returns Previous value of *target*.

static inline atomic_ptr_val_t `atomic_ptr_set`(atomic_ptr_t *target, atomic_ptr_val_t value)

Atomic get-and-set for pointer values.

This routine atomically sets *target* to *value* and returns the previous value of *target*.

Parameters

- `target` – Address of atomic variable.
- `value` – Value to write to *target*.

Returns Previous value of *target*.

static inline atomic_val_t `atomic_clear`(atomic_t *target)

Atomic clear.

This routine atomically sets *target* to zero and returns its previous value. (Hence, it is equivalent to atomic_set(target, 0).)

Parameters

- `target` – Address of atomic variable.

Returns Previous value of *target*.

static inline atomic_ptr_val_t `atomic_ptr_clear`(atomic_ptr_t *target)

Atomic clear of a pointer value.

This routine atomically sets *target* to zero and returns its previous value. (Hence, it is equivalent to atomic_set(target, 0).)

Parameters

- `target` – Address of atomic variable.

Returns Previous value of *target*.

static inline atomic_val_t `atomic_or`(atomic_t *target, atomic_val_t value)

Atomic bitwise inclusive OR.

This routine atomically sets *target* to the bitwise inclusive OR of *target* and *value*.

Parameters

- `target` – Address of atomic variable.
- `value` – Value to OR.

Returns Previous value of *target*.

static inline atomic_val_t `atomic_xor`(atomic_t *target, atomic_val_t value)

Atomic bitwise exclusive OR (XOR).

This routine atomically sets *target* to the bitwise exclusive OR (XOR) of *target* and *value*.

Parameters

- `target` – Address of atomic variable.
- `value` – Value to XOR

Returns Previous value of *target*.

static inline atomic_val_t `atomic_and`(atomic_t *target, atomic_val_t value)

Atomic bitwise AND.

This routine atomically sets *target* to the bitwise AND of *target* and *value*.

Parameters

- `target` – Address of atomic variable.
- `value` – Value to AND.

Returns Previous value of *target*.

static inline atomic_val_t `atomic_nand`(atomic_t *target, atomic_val_t value)

Atomic bitwise NAND.

This routine atomically sets *target* to the bitwise NAND of *target* and *value*. (This operation is equivalent to target = ~(target & value).)

Parameters

- `target` – Address of atomic variable.

- `value` – Value to NAND.

Returns Previous value of *target*.

Floating Point Services

The kernel allows threads to use floating point registers on board configurations that support these registers.

Note: Floating point services are currently available only for boards based on ARM Cortex-M SoCs supporting the Floating Point Extension, the Intel x86 architecture, the SPARC architecture and ARCv2 SoCs supporting the Floating Point Extension. The services provided are architecture specific.

The kernel does not support the use of floating point registers by ISRs.

Concepts The kernel can be configured to provide only the floating point services required by an application. Three modes of operation are supported, which are described below. In addition, the kernel's support for the SSE registers can be included or omitted, as desired.

No FP registers mode This mode is used when the application has no threads that use floating point registers. It is the kernel's default floating point services mode.

If a thread uses any floating point register, the kernel generates a fatal error condition and aborts the thread.

Unshared FP registers mode This mode is used when the application has only a single thread that uses floating point registers.

On x86 platforms, the kernel initializes the floating point registers so they can be used by any thread (initialization in skipped on ARM Cortex-M platforms and ARCv2 platforms). The floating point registers are left unchanged whenever a context switch occurs.

Note: The behavior is undefined, if two or more threads attempt to use the floating point registers, as the kernel does not attempt to detect (or prevent) multiple threads from using these registers.

Shared FP registers mode This mode is used when the application has two or more threads that use floating point registers. Depending upon the underlying CPU architecture, the kernel supports one or more of the following thread sub-classes:

- non-user: A thread that cannot use any floating point registers
- FPU user: A thread that can use the standard floating point registers
- SSE user: A thread that can use both the standard floating point registers and SSE registers

The kernel initializes and enables access to the floating point registers, so they can be used by any thread, then saves and restores these registers during context switches to ensure the computations performed by each FPU user or SSE user are not impacted by the computations performed by the other users.

ARM Cortex-M architecture (with the Floating Point Extension)

Note: The Shared FP registers mode is the default Floating Point Services mode in ARM Cortex-M.

On the ARM Cortex-M architecture with the Floating Point Extension, the kernel treats *all* threads as FPU users when shared FP registers mode is enabled. This means that any thread is allowed to access the floating point registers. The ARM kernel automatically detects that a given thread is using the floating point registers the first time the thread accesses them.

Pretag a thread that intends to use the FP registers by using one of the techniques listed below.

- A statically-created ARM thread can be pretagged by passing the *K_FP_REGS* option to *K_THREAD_DEFINE*.
- A dynamically-created ARM thread can be pretagged by passing the *K_FP_REGS* option to *k_thread_create()*.

Pretagging a thread with the *K_FP_REGS* option instructs the MPU-based stack protection mechanism to properly configure the size of the thread's guard region to always guarantee stack overflow detection, and enable lazy stacking for the given thread upon thread creation.

During thread context switching the ARM kernel saves the *callee-saved* floating point registers, if the switched-out thread has been using them. Additionally, the *caller-saved* floating point registers are saved on the thread's stack. If the switched-in thread has been using the floating point registers, the kernel restores the *callee-saved* FP registers of the switched-in thread and the *caller-saved* FP context is restored from the thread's stack. Thus, the kernel does not save or restore the FP context of threads that are not using the FP registers.

Each thread that intends to use the floating point registers must provide an extra 72 bytes of stack space where the callee-saved FP context can be saved.

Lazy Stacking is currently enabled in Zephyr applications on ARM Cortex-M architecture, minimizing interrupt latency, when the floating point context is active.

When the MPU-based stack protection mechanism is not enabled, lazy stacking is always active in the Zephyr application. When the MPU-based stack protection is enabled, the following rules apply with respect to lazy stacking:

- Lazy stacking is activated by default on threads that are pretagged with *K_FP_REGS*
- Lazy stacking is activated dynamically on threads that are not pretagged with *K_FP_REGS*, as soon as the kernel detects that they are using the floating point registers.

If an ARM thread does not require use of the floating point registers any more, it can call `k_float_disable()`. This instructs the kernel not to save or restore its FP context during thread context switching.

ARM64 architecture

Note: The Shared FP registers mode is the default Floating Point Services mode on ARM64. The compiler is free to optimize code using FP/SIMD registers, and library functions such as memcpy are known to make use of them.

On the ARM64 (Aarch64) architecture the kernel treats each thread as a FPU user on a case-by-case basis. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the registers are *not* saved when switching from an FPU user to a non-user thread, and then back to the original FPU user.

FPU register usage by ISRs is supported although not recommended. When an ISR uses floating point or SIMD registers, then the access is trapped, the current FPU user context is saved in the thread object and the ISR is resumed with interrupts disabled so to prevent another IRQ from interrupting the ISR and potentially requesting FPU usage. Because ISR don't have a persistent register context, there are no provision for saving an ISR's FPU context either, hence the IRQ disabling.

Each thread object becomes 512 bytes larger when Shared FP registers mode is enabled.

ARCv2 architecture On the ARCv2 architecture, the kernel treats each thread as a non-user or FPU user and the thread must be tagged by one of the following techniques.

- A statically-created ARC thread can be tagged by passing the *K_FP_REGS* option to *K_THREAD_DEFINE*.
- A dynamically-created ARC thread can be tagged by passing the *K_FP_REGS* to *k_thread_create()*.

If an ARC thread does not require use of the floating point registers any more, it can call `k_float_disable()`. This instructs the kernel not to save or restore its FP context during thread context switching.

During thread context switching the ARC kernel saves the *callee-saved* floating point registers, if the switched-out thread has been using them. Additionally, the *caller-saved* floating point registers are saved on the thread's stack. If the switched-in thread has been using the floating point registers, the kernel restores the *callee-saved* FP registers of the switched-in thread and the *caller-saved* FP context is restored from the thread's stack. Thus, the kernel does not save or restore the FP context of threads that are not using the FP registers. An extra 16 bytes (single floating point hardware) or 32 bytes (double floating point hardware) of stack space is required to load and store floating point registers.

RISC-V architecture On the RISC-V architecture, the kernel treats each thread as a non-user or FPU user and the thread must be tagged by one of the following techniques:

- A statically-created RISC-V thread can be tagged by passing the *K_FP_REGS* option to *K_THREAD_DEFINE*.
- A dynamically-created RISC-V thread can be tagged by passing the *K_FP_REGS* to *k_thread_create()*.
- A running RISC-V thread can be tagged by calling `k_float_enable()`. This function can only be called from the thread itself.

If a RISC-V thread no longer requires the use of the floating point registers, it can call `k_float_disable()`. This instructs the kernel not to save or restore its FP context during thread context switching. This function can only be called from the thread itself.

During thread context switching the RISC-V kernel saves the *callee-saved* floating point registers, if the switched-out thread is tagged with *K_FP_REGS*. Additionally, the *caller-saved* floating point registers are saved on the thread's stack. If the switched-in thread has been tagged with *K_FP_REGS*, then the kernel restores the *callee-saved* FP registers of the switched-in thread and the *caller-saved* FP context is restored from the thread's stack. Thus, the kernel does not save or restore the FP context of threads that are not using the FP registers. An extra 84 bytes (single floating point hardware) or 164 bytes (double floating point hardware) of stack space is required to load and store floating point registers.

SPARC architecture On the SPARC architecture, the kernel treats each thread as a non-user or FPU user and the thread must be tagged by one of the following techniques:

- A statically-created thread can be tagged by passing the *K_FP_REGS* option to *K_THREAD_DEFINE*.
- A dynamically-created thread can be tagged by passing the *K_FP_REGS* to *k_thread_create()*.

During thread context switch at exit from interrupt handler, the SPARC kernel saves *all* floating point registers, if the FPU was enabled in the switched-out thread. Floating point registers are saved on the thread's stack. Floating point registers are restored when a thread context is restored iff they were saved at the context save. Saving and restoring of the floating point registers is synchronous and thus not lazy. The FPU is always disabled when an ISR is called (independent of `CONFIG_FPU_SHARING`).

Floating point disabling with `k_float_disable()` is not implemented.

When `CONFIG_FPU_SHARING` is used, then 136 bytes of stack space is required for each FPU user thread to load and store floating point registers. No extra stack is required if `CONFIG_FPU_SHARING` is not used.

x86 architecture On the x86 architecture the kernel treats each thread as a non-user, FPU user or SSE user on a case-by-case basis. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the registers are *not* saved when switching from an FPU user to a non-user thread, and then back to the original FPU user. The following table indicates the amount of additional stack space a thread must provide so the registers can be saved properly.

Thread type	FP register use	Extra stack space required
cooperative	any	0 bytes
preemptive	none	0 bytes
preemptive	FPU	108 bytes
preemptive	SSE	464 bytes

The x86 kernel automatically detects that a given thread is using the floating point registers the first time the thread accesses them. The thread is tagged as an SSE user if the kernel has been configured to support the SSE registers, or as an FPU user if the SSE registers are not supported. If this would result in a thread that is an FPU user being tagged as an SSE user, or if the application wants to avoid the exception handling overhead involved in auto-tagging threads, it is possible to pretag a thread using one of the techniques listed below.

- A statically-created x86 thread can be pretagged by passing the *K_FP_REGS* or `K_SSE_REGS` option to *K_THREAD_DEFINE*.
- A dynamically-created x86 thread can be pretagged by passing the *K_FP_REGS* or `K_SSE_REGS` option to *k_thread_create()*.
- An already-created x86 thread can pretag itself once it has started by passing the *K_FP_REGS* or `K_SSE_REGS` option to `k_float_enable()`.

If an x86 thread uses the floating point registers infrequently it can call `k_float_disable()` to remove its tagging as an FPU user or SSE user. This eliminates the need for the kernel to take steps to preserve the contents of the floating point registers during context switches when there is no need to do so. When the thread again needs to use the floating point registers it can re-tag itself as an FPU user or SSE user by calling `k_float_enable()`.

Implementation

Performing Floating Point Arithmetic No special coding is required for a thread to use floating point arithmetic if the kernel is properly configured.

The following code shows how a routine can use floating point arithmetic to avoid overflow issues when computing the average of a series of integer values.

```
int average(int *values, int num_values)
{
    double sum;
    int i;

    sum = 0.0;

    for (i = 0; i < num_values; i++) {
        sum += *values;
        values++;
    }

    return (int)((sum / num_values) + 0.5);
}
```

Suggested Uses Use the kernel floating point services when an application needs to perform floating point operations.

Configuration Options To configure unshared FP registers mode, enable the `CONFIG_FPU` configuration option and leave the `CONFIG_FPU_SHARING` configuration option disabled.

To configure shared FP registers mode, enable both the `CONFIG_FPU` configuration option and the `CONFIG_FPU_SHARING` configuration option. Also, ensure that any thread that uses the floating point registers has sufficient added stack space for saving floating point register values during context switches, as described above.

For x86, use the `CONFIG_X86_SSE` configuration option to enable support for SSEx instructions.

API Reference

group `float_apis`

C++ Support for Applications

The kernel supports applications written in both C and C++. However, to use C++ in an application you must configure the kernel to include C++ support and the build system must select the correct compiler.

The build system selects the C++ compiler based on the suffix of the files. Files identified with either a **cxx** or a **cpp** suffix are compiled using the C++ compiler. For example, `myCplusplusApp.cpp` is compiled using C++.

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

- Dynamic object management with the **new** and **delete** operators
- RTTI (runtime type information)
- Static global object destruction

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

- Inheritance
- Virtual functions
- Virtual tables
- Static global object constructors

- Exceptions

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

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

Version

Kernel version handling and APIs related to kernel version being used.

API Reference

uint32_t `sys_kernel_version_get`(void)

Return the kernel version of the present build.

The kernel version is a four-byte value, whose format is described in the file "kernel_version.h".

Returns kernel version

`SYS_KERNEL_VER_MAJOR`(ver)

`SYS_KERNEL_VER_MINOR`(ver)

`SYS_KERNEL_VER_PATCHLEVEL`(ver)

Fatal Errors

Software Errors Triggered in Source Code Zephyr provides several methods for inducing fatal error conditions through either build-time checks, conditionally compiled assertions, or deliberately invoked panic or oops conditions.

Runtime Assertions Zephyr provides some macros to perform runtime assertions which may be conditionally compiled. Their definitions may be found in include/sys/__assert.h.

Assertions are enabled by setting the `__ASSERT_ON` preprocessor symbol to a non-zero value. There are two ways to do this:

- Use the `CONFIG_ASSERT` and `CONFIG_ASSERT_LEVEL` kconfig options.
- Add `-D__ASSERT_ON=<level>` to the project's CFLAGS, either on the build command line or in a CMakeLists.txt.

The `__ASSERT_ON` method takes precedence over the kconfig option if both are used.

Specifying an assertion level of 1 causes the compiler to issue warnings that the kernel contains debug-type `__ASSERT()` statements; this reminder is issued since assertion code is not normally present in a final product. Specifying assertion level 2 suppresses these warnings.

Assertions are enabled by default when running Zephyr test cases, as configured by the `CONFIG_TEST` option.

The policy for what to do when encountering a failed assertion is controlled by the implementation of `assert_post_action()`. Zephyr provides a default implementation with weak linkage which invokes a kernel oops if the thread that failed the assertion was running in user mode, and a kernel panic otherwise.

__ASSERT() The `__ASSERT()` macro can be used inside kernel and application code to perform optional runtime checks which will induce a fatal error if the check does not pass. The macro takes a string message which will be printed to provide context to the assertion. In addition, the kernel will print a text representation of the expression code that was evaluated, and the file and line number where the assertion can be found.

For example:

```
__ASSERT(foo == 0xF0CACC1A, "Invalid value of foo, got 0x%x", foo);
```

If at runtime `foo` had some unexpected value, the error produced may look like the following:

```
ASSERTION FAIL [foo == 0xF0CACC1A] @ ZEPHYR_BASE/tests/kernel/fatal/src/main.c:367
        Invalid value of foo, got 0xdeadbeef
[00:00:00.000,000] <err> os: r0/a1:  0x00000004  r1/a2:  0x0000016f  r2/a3: ␣
↪0x00000000
[00:00:00.000,000] <err> os: r3/a4:  0x00000000 r12/ip:  0x00000000 r14/lr: ␣
↪0x00000a6d
[00:00:00.000,000] <err> os:  xpsr:  0x61000000
[00:00:00.000,000] <err> os: Faulting instruction address (r15/pc): 0x00009fe4
[00:00:00.000,000] <err> os: >>> ZEPHYR FATAL ERROR 4: Kernel panic
[00:00:00.000,000] <err> os: Current thread: 0x20000414 (main)
[00:00:00.000,000] <err> os: Halting system
```

__ASSERT_EVAL() The `__ASSERT_EVAL()` macro can also be used inside kernel and application code, with special semantics for the evaluation of its arguments.

It makes use of the `__ASSERT()` macro, but has some extra flexibility. It allows the developer to specify different actions depending whether the `__ASSERT()` macro is enabled or not. This can be particularly useful to prevent the compiler from generating comments (errors, warnings or remarks) about variables that are only used with `__ASSERT()` being assigned a value, but otherwise unused when the `__ASSERT()` macro is disabled.

Consider the following example:

```
int x;
x = foo();
__ASSERT(x != 0, "foo() returned zero!");
```

If `__ASSERT()` is disabled, then 'x' is assigned a value, but never used. This type of situation can be resolved using the __ASSERT_EVAL() macro.

```
__ASSERT_EVAL ((void) foo(),
               int x = foo(),
               x != 0,
               "foo() returned zero!");
```

The first parameter tells `__ASSERT_EVAL()` what to do if `__ASSERT()` is disabled. The second parameter tells `__ASSERT_EVAL()` what to do if `__ASSERT()` is enabled. The third and fourth parameters are the parameters it passes to `__ASSERT()`.

__ASSERT_NO_MSG() The `__ASSERT_NO_MSG()` macro can be used to perform an assertion that reports the failed test and its location, but lacks additional debugging information provided to assist the user in diagnosing the problem; its use is discouraged.

Build Assertions Zephyr provides two macros for performing build-time assertion checks. These are evaluated completely at compile-time, and are always checked.

BUILD_ASSERT() This has the same semantics as C's `_Static_assert` or C++'s `static_assert`. If the evaluation fails, a build error will be generated by the compiler. If the compiler supports it, the provided message will be printed to provide further context.

Unlike `__ASSERT()`, the message must be a static string, without `printf()`-like format codes or extra arguments.

For example, suppose this check fails:

```
BUILD_ASSERT(FOO == 2000, "Invalid value of FOO");
```

With GCC, the output resembles:

```
tests/kernel/fatal/src/main.c: In function 'test_main':
include/toolchain/gcc.h:28:37: error: static assertion failed: "Invalid value of FOO"
 #define BUILD_ASSERT(EXPR, MSG) _Static_assert(EXPR, "" MSG)
                                 ^~~~~~~~~~~~~~
tests/kernel/fatal/src/main.c:370:2: note: in expansion of macro 'BUILD_ASSERT'
  BUILD_ASSERT(FOO == 2000,
  ^~~~~~~~~~~~~~~~
```

Kernel Oops A kernel oops is a software triggered fatal error invoked by `k_oops()`. This should be used to indicate an unrecoverable condition in application logic.

The fatal error reason code generated will be `K_ERR_KERNEL_OOPS`.

Kernel Panic A kernel error is a software triggered fatal error invoked by `k_panic()`. This should be used to indicate that the Zephyr kernel is in an unrecoverable state. Implementations of *`k_sys_fatal_error_handler()`* should not return if the kernel encounters a panic condition, as the entire system needs to be reset.

Threads running in user mode are not permitted to invoke `k_panic()`, and doing so will generate a kernel oops instead. Otherwise, the fatal error reason code generated will be `K_ERR_KERNEL_PANIC`.

Exceptions

Spurious Interrupts If the CPU receives a hardware interrupt on an interrupt line that has not had a handler installed with `IRQ_CONNECT()` or *`irq_connect_dynamic()`*, then the kernel will generate a fatal error with the reason code `K_ERR_SPURIOUS_IRQ()`.

Stack Overflows In the event that a thread pushes more data onto its execution stack than its stack buffer provides, the kernel may be able to detect this situation and generate a fatal error with a reason code of `K_ERR_STACK_CHK_FAIL`.

If a thread is running in user mode, then stack overflows are always caught, as the thread will simply not have permission to write to adjacent memory addresses outside of the stack buffer. Because this is enforced by the memory protection hardware, there is no risk of data corruption to memory that the thread would not otherwise be able to write to.

If a thread is running in supervisor mode, or if `CONFIG_USERSPACE` is not enabled, depending on configuration stack overflows may or may not be caught. `CONFIG_HW_STACK_PROTECTION` is supported on some architectures and will catch stack overflows in supervisor mode, including when handling a system call on behalf of a user thread. Typically this is implemented via dedicated CPU features, or read-only MMU/MPU guard regions placed immediately adjacent to the stack buffer. Stack overflows caught in this way can detect the overflow, but cannot guarantee against data corruption and should be treated as a very serious condition impacting the health of the entire system.

If a platform lacks memory management hardware support, CONFIG_STACK_SENTINEL is a software-only stack overflow detection feature which periodically checks if a sentinel value at the end of the stack buffer has been corrupted. It does not require hardware support, but provides no protection against data corruption. Since the checks are typically done at interrupt exit, the overflow may be detected a nontrivial amount of time after the stack actually overflowed.

Finally, Zephyr supports GCC compiler stack canaries via CONFIG_STACK_CANARIES. If enabled, the compiler will insert a canary value randomly generated at boot into function stack frames, checking that the canary has not been overwritten at function exit. If the check fails, the compiler invokes __stack_chk_fail(), whose Zephyr implementation invokes a fatal stack overflow error. An error in this case does not indicate that the entire stack buffer has overflowed, but instead that the current function stack frame has been corrupted. See the compiler documentation for more details.

Other Exceptions Any other type of unhandled CPU exception will generate an error code of K_ERR_CPU_EXCEPTION.

Fatal Error Handling The policy for what to do when encountering a fatal error is determined by the implementation of the *k_sys_fatal_error_handler()* function. This function has a default implementation with weak linkage that calls LOG_PANIC() to dump all pending logging messages and then unconditionally halts the system with *k_fatal_halt()*.

Applications are free to implement their own error handling policy by overriding the implementation of *k_sys_fatal_error_handler()*. If the implementation returns, the faulting thread will be aborted and the system will otherwise continue to function. See the documentation for this function for additional details and constraints.

API Reference

group fatal_apis

Enums

enum k_fatal_error_reason

Values:

enumerator K_ERR_CPU_EXCEPTION

Generic CPU exception, not covered by other codes

enumerator K_ERR_SPURIOUS_IRQ

Unhandled hardware interrupt

enumerator K_ERR_STACK_CHK_FAIL

Faulting context overflowed its stack buffer

enumerator K_ERR_KERNEL_OOPS

Moderate severity software error

enumerator K_ERR_KERNEL_PANIC

High severity software error

Functions

FUNC_NORETURN void `k_fatal_halt`(unsigned int reason)

Halt the system on a fatal error.

Invokes architecture-specific code to power off or halt the system in a low power state. Lacking that, lock interrupts and sit in an idle loop.

Parameters

- `reason` – Fatal exception reason code

void `k_sys_fatal_error_handler`(unsigned int reason, const z_arch_esf_t *esf)

Fatal error policy handler.

This function is not invoked by application code, but is declared as a weak symbol so that applications may introduce their own policy.

The default implementation of this function halts the system unconditionally. Depending on architecture support, this may be a simple infinite loop, power off the hardware, or exit an emulator.

If this function returns, then the currently executing thread will be aborted.

A few notes for custom implementations:

- If the error is determined to be unrecoverable, *LOG_PANIC()* should be invoked to flush any pending logging buffers.
- K_ERR_KERNEL_PANIC indicates a severe unrecoverable error in the kernel itself, and should not be considered recoverable. There is an assertion in z_fatal_error() to enforce this.
- Even outside of a kernel panic, unless the fault occurred in user mode, the kernel itself may be in an inconsistent state, with API calls to kernel objects possibly exhibiting undefined behavior or triggering another exception.

Parameters

- `reason` – The reason for the fatal error
- `esf` – Exception context, with details and partial or full register state when the error occurred. May in some cases be NULL.

Thread Local Storage (TLS)

Thread Local Storage (TLS) allows variables to be allocated on a per-thread basis. These variables are stored in the thread stack which means every thread has its own copy of these variables.

Zephyr currently requires toolchain support for TLS.

Configuration To enable thread local storage in Zephyr, `CONFIG_THREAD_LOCAL_STORAGE` needs to be enabled. Note that this option may not be available if the architecture or the SoC does not have the hidden option `CONFIG_ARCH_HAS_THREAD_LOCAL_STORAGE` enabled, which means the architecture or the SoC does not have the necessary code to support thread local storage and/or the toolchain does not support TLS.

`CONFIG_ERRNO_IN_TLS` can be enabled together with `CONFIG_ERRNO` to let the variable `errno` be a thread local variable. This allows user threads to access the value of `errno` without making a system call.

Declaring and Using Thread Local Variables The keyword `__thread` can be used to declare thread local variables.

For example, to declare a thread local variable in header files:

```
extern __thread int i;
```

And to declare the actual variable in source files:

```
__thread int i;
```

Keyword `static` can also be used to limit the variable within a source file:

```
static __thread int j;
```

Using the thread local variable is the same as using other variable, for example:

```
void testing(void) {
    i = 10;
}
```

7.14 C standard library

The C standard library is an integral part of any C program, and Zephyr provides two implementations for the application to choose from.

The first one, named "minimal libc" is part of the Zephyr code base and provides the minimal subset of the standard C library required to meet the needs of Zephyr and its subsystems and features, primarily in the areas of string manipulation and display. It is very low footprint and is suitable for projects that do not rely on less frequently used portions of the ISO C standard library. Its implementation can be found in `lib/libc/minimal` in the main zephyr tree.

The second one is newlib, a complete C library implementation written for embedded systems. Newlib is separate open source project and is not included in source code form with Zephyr. Instead, the *Install the Zephyr Software Development Kit (SDK)* comes with a precompiled library for each supported architecture (`libc.a` and `libm.a`). Other 3rd-party toolchains, such as *GNU Arm Embedded*, also bundle newlib as a precompiled library. Newlib can be enabled by selecting the `CONFIG_NEWLIB_LIBC` in the application configuration file. Part of the support for `newlib` is a set of hooks available under `lib/libc/newlib/libc-hooks.c` which integrates the C standard library with basic kernel services.

7.14.1 API Reference

Error numbers

Error numbers are used throughout Zephyr APIs to signal error conditions as return values from functions. They are typically returned as the negative value of the integer literals defined in this section, and are defined in the `errno.h` header file. A subset of the error numbers are defined in the POSIX errno.h specification, and others have been added to it from other sources.

A conscious effort is made in Zephyr to keep the values of system error numbers consistent between the different implementations of the C standard library. The version of `errno.h` that is in the main zephyr tree, errno.h, is checked against newlib's own list to ensure that the error numbers are kept aligned.

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

group system_errno

System error numbers Error codes returned by functions. Includes a list of those defined by IEEE Std 1003.1-2017.

Defines

errno

EPERM
: Not owner

ENOENT
: No such file or directory

ESRCH
: No such context

EINTR
: Interrupted system call

EIO
: I/O error

ENXIO
: No such device or address

E2BIG
: Arg list too long

ENOEXEC
: Exec format error

EBADF
: Bad file number

ECHILD
: No children

EAGAIN
: No more contexts

ENOMEM
: Not enough core

EACCES
: Permission denied

EFAULT
: Bad address

ENOTBLK
: Block device required

EBUSY
: Mount device busy

EEXIST
: File exists

EXDEV
: Cross-device link

ENODEV
: No such device

ENOTDIR
: Not a directory

EISDIR
: Is a directory

EINVAL
: Invalid argument

ENFILE
: File table overflow

EMFILE
: Too many open files

ENOTTY
: Not a typewriter

ETXTBSY
: Text file busy

EFBIG
: File too large

ENOSPC
: No space left on device

ESPIPE
: Illegal seek

EROFS
: Read-only file system

EMLINK
: Too many links

EPIPE
: Broken pipe

EDOM
: Argument too large

ERANGE
: Result too large

ENOMSG
: Unexpected message type

EDEADLK
: Resource deadlock avoided

ENOLCK
: No locks available

ENOSTR
: STREAMS device required

ENODATA
: Missing expected message data

ETIME
: STREAMS timeout occurred

ENOSR
: Insufficient memory

EPROTO
: Generic STREAMS error

EBADMSG
: Invalid STREAMS message

ENOSYS
: Function not implemented

ENOTEMPTY
: Directory not empty

ENAMETOOLONG
: File name too long

ELOOP
: Too many levels of symbolic links

EOPNOTSUPP
: Operation not supported on socket

EPFNOSUPPORT
: Protocol family not supported

ECONNRESET
: Connection reset by peer

ENOBUFS
: No buffer space available

EAFNOSUPPORT
: Addr family not supported

EPROTOTYPE
: Protocol wrong type for socket

ENOTSOCK
: Socket operation on non-socket

ENOPROTOOPT
: Protocol not available

ESHUTDOWN
: Can't send after socket shutdown

ECONNREFUSED
: Connection refused

EADDRINUSE
: Address already in use

ECONNABORTED
: Software caused connection abort

ENETUNREACH
: Network is unreachable

ENETDOWN
: Network is down

ETIMEDOUT
: Connection timed out

EHOSTDOWN
: Host is down

EHOSTUNREACH
: No route to host

EINPROGRESS
: Operation now in progress

EALREADY
: Operation already in progress

EDESTADDRREQ
: Destination address required

EMSGSIZE
: Message size

EPROTONOSUPPORT
: Protocol not supported

ESOCKTNOSUPPORT
: Socket type not supported

EADDRNOTAVAIL
: Can't assign requested address

ENETRESET
: Network dropped connection on reset

EISCONN
: Socket is already connected

ENOTCONN
: Socket is not connected

ETOOMANYREFS
: Too many references: can't splice

ENOTSUP
: Unsupported value

EILSEQ
: Illegal byte sequence

`EOVERFLOW`
: Value overflow

`ECANCELED`
: Operation canceled

`EWOULDBLOCK`
: Operation would block

7.15 Logging

The logging API provides a common interface to process messages issued by developers. Messages are passed through a frontend and are then processed by active backends. Custom frontend and backends can be used if needed. Default configuration uses built-in frontend and UART backend.

Summary of the logging features:

- Deferred logging reduces the time needed to log a message by shifting time consuming operations to a known context instead of processing and sending the log message when called.
- Multiple backends supported (up to 9 backends).

- Custom frontend supported.
- Compile time filtering on module level.
- Run time filtering independent for each backend.
- Additional run time filtering on module instance level.
- Timestamping with user provided function.
- Dedicated API for dumping data.
- Dedicated API for handling transient strings.
- Panic support - in panic mode logging switches to blocking, synchronous processing.
- Printk support - printk message can be redirected to the logging.
- Design ready for multi-domain/multi-processor system.

Logging v2 introduces following changes:

- Option to use 64 bit timestamp
- Support for logging floating point variables
- Support for logging variables extending size of a machine word (64 bit values on 32 bit architectures)
- Remove the need for special treatment of `%s` format specifier
- Extend API for dumping data to accept formatted string
- Improve memory utilization. More log messages fit in the logging buffer in deferred mode.
- Log message is no longer fragmented. It is self-contained block of memory which simplifies out of domain handling (e.g. offline processing)
- Improved performance when logging from user space
- Improved performance when logging to full buffer and message are dropped.
- Slightly degrade performance in normal circumstances due to the fact that allocation from ring buffer is more complex than from memslab.
- No change in logging API
- Logging backend API exteded with function for processing v2 messages.

Logging API is highly configurable at compile time as well as at run time. Using Kconfig options (see *Global Kconfig Options*) logs can be gradually removed from compilation to reduce image size and execution time when logs are not needed. During compilation logs can be filtered out on module basis and severity level.

Logs can also be compiled in but filtered on run time using dedicate API. Run time filtering is independent for each backend and each source of log messages. Source of log messages can be a module or specific instance of the module.

There are four severity levels available in the system: error, warning, info and debug. For each severity level the logging API (include/logging/log.h) has set of dedicated macros. Logger API also has macros for logging data.

For each level following set of macros are available:

- `LOG_X` for standard printf-like messages, e.g. *`LOG_ERR`*.
- `LOG_HEXDUMP_X` for dumping data, e.g. *`LOG_HEXDUMP_WRN`*.
- `LOG_INST_X` for standard printf-like message associated with the particular instance, e.g. *`LOG_INST_INF`*.
- `LOG_INST_HEXDUMP_X` for dumping data associated with the particular instance, e.g. `LOG_HEXDUMP_INST_DBG`

There are two configuration categories: configurations per module and global configuration. When logging is enabled globally, it works for modules. However, modules can disable logging locally. Every module can specify its own logging level. The module must define the LOG_LEVEL macro before using the API. Unless a global override is set, the module logging level will be honored. The global override can only increase the logging level. It cannot be used to lower module logging levels that were previously set higher. It is also possible to globally limit logs by providing maximal severity level present in the system, where maximal means lowest severity (e.g. if maximal level in the system is set to info, it means that errors, warnings and info levels are present but debug messages are excluded).

Each module which is using the logging must specify its unique name and register itself to the logging. If module consists of more than one file, registration is performed in one file but each file must define a module name.

Logger's default frontend is designed to be thread safe and minimizes time needed to log the message. Time consuming operations like string formatting or access to the transport are not performed by default when logging API is called. When logging API is called a message is created and added to the list. Dedicated, configurable buffer for pool of log messages is used. There are 2 types of messages: standard and hexdump. Each message contain source ID (module or instance ID and domain ID which might be used for multiprocessor systems), timestamp and severity level. Standard message contains pointer to the string and arguments. Hexdump message contains copied data and string.

7.15.1 Global Kconfig Options

These options can be found in the following path subsys/logging/Kconfig.

CONFIG_LOG: Global switch, turns on/off the logging.

Mode of operations:

CONFIG_LOG_MODE_DEFERRED: Deferred mode.

CONFIG_LOG2_MODE_DEFERRED: Deferred mode v2.

CONFIG_LOG_MODE_IMMEDIATE: Immediate (synchronous) mode.

CONFIG_LOG2_MODE_IMMEDIATE: Immediate (synchronous) mode v2.

CONFIG_LOG_MODE_MINIMAL: Minimal footprint mode.

Filtering options:

CONFIG_LOG_RUNTIME_FILTERING: Enables runtime reconfiguration of the filtering.

CONFIG_LOG_DEFAULT_LEVEL: Default level, sets the logging level used by modules that are not setting their own logging level.

CONFIG_LOG_OVERRIDE_LEVEL: It overrides module logging level when it is not set or set lower than the override value.

CONFIG_LOG_MAX_LEVEL: Maximal (lowest severity) level which is compiled in.

Processing options:

CONFIG_LOG_MODE_OVERFLOW: When new message cannot be allocated, oldest one are discarded.

CONFIG_LOG_BLOCK_IN_THREAD: If enabled and new log message cannot be allocated thread context will block for up to CONFIG_LOG_BLOCK_IN_THREAD_TIMEOUT_MS or until log message is allocated.

CONFIG_LOG_PRINTK: Redirect printk calls to the logging.

CONFIG_LOG_PRINTK_MAX_STRING_LENGTH: Maximal string length that can be processed by printk. Longer strings are trimmed.

CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD: When number of buffered log messages reaches the threshold dedicated thread (see *log_thread_set()*) is waken up. If CONFIG_LOG_PROCESS_THREAD is enabled then this threshold is used by the internal thread.

CONFIG_LOG_PROCESS_THREAD: When enabled, logging thread is created which handles log processing.

CONFIG_LOG_PROCESS_THREAD_STARTUP_DELAY_MS: Delay in milliseconds after which logging thread is started.

CONFIG_LOG_BUFFER_SIZE: Number of bytes dedicated for the message pool. Single message capable of storing standard log with up to 3 arguments or hexdump message with 12 bytes of data take 32 bytes. In v2 it indicates buffer size dedicated for circular packet buffer.

CONFIG_LOG_DETECT_MISSED_STRDUP: Enable detection of missed transient strings handling.

CONFIG_LOG_STRDUP_MAX_STRING: Longest string that can be duplicated using log_strdup().

CONFIG_LOG_STRDUP_BUF_COUNT: Number of buffers in the pool used by log_strdup().

CONFIG_LOG_DOMAIN_ID: Domain ID. Valid in multi-domain systems.

CONFIG_LOG_FRONTEND: Redirect logs to a custom frontend.

CONFIG_LOG_TIMESTAMP_64BIT: 64 bit timestamp.

Formatting options:

CONFIG_LOG_FUNC_NAME_PREFIX_ERR: Prepend standard ERROR log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_WRN: Prepend standard WARNING log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_INF: Prepend standard INFO log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_DBG: Prepend standard DEBUG log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_BACKEND_SHOW_COLOR: Enables coloring of errors (red) and warnings (yellow).

CONFIG_LOG_BACKEND_FORMAT_TIMESTAMP: If enabled timestamp is formatted to *hh:mm:ss:mmm,uuu*. Otherwise is printed in raw format.

Backend options:

CONFIG_LOG_BACKEND_UART: Enabled build-in UART backend.

7.15.2 Usage

Logging in a module

In order to use logging in the module, a unique name of a module must be specified and module must be registered using *LOG_MODULE_REGISTER*. Optionally, a compile time log level for the module can be specified as the second parameter. Default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if custom log level is not provided.

```
#include <logging/log.h>
LOG_MODULE_REGISTER(foo, CONFIG_FOO_LOG_LEVEL);
```

If the module consists of multiple files, then LOG_MODULE_REGISTER() should appear in exactly one of them. Each other file should use *LOG_MODULE_DECLARE* to declare its membership in the module. Optionally, a compile time log level for the module can be specified as the second parameter. Default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if custom log level is not provided.

```
#include <logging/log.h>
/* In all files comprising the module but one */
LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL);
```

In order to use logging API in a function implemented in a header file *LOG_MODULE_DECLARE* macro must be used in the function body before logging API is called. Optionally, a compile time log level for the module can be specified as the second parameter. Default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if custom log level is not provided.

```
#include <logging/log.h>

static inline void foo(void)
{
     LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL);

     LOG_INF("foo");
}
```

Dedicated Kconfig template (subsys/logging/Kconfig.template.log_config) can be used to create local log level configuration.

Example below presents usage of the template. As a result CONFIG_FOO_LOG_LEVEL will be generated:

```
module = FOO
module-str = foo
source "subsys/logging/Kconfig.template.log_config"
```

Logging in a module instance

In case of modules which are multi-instance and instances are widely used across the system enabling logs will lead to flooding. Logger provide the tools which can be used to provide filtering on instance level rather than module level. In that case logging can be enabled for particular instance.

In order to use instance level filtering following steps must be performed:

- a pointer to specific logging structure is declared in instance structure. LOG_INSTANCE_PTR_DECLARE is used for that.

```
#include <logging/log_instance.h>

struct foo_object {
     LOG_INSTANCE_PTR_DECLARE(log);
     uint32_t id;
}
```

- module must provide macro for instantiation. In that macro, logging instance is registered and log instance pointer is initialized in the object structure.

```
#define FOO_OBJECT_DEFINE(_name)                                \
     LOG_INSTANCE_REGISTER(foo, _name, CONFIG_FOO_LOG_LEVEL) \
     struct foo_object _name = {                                \
             LOG_INSTANCE_PTR_INIT(log, foo, _name)          \
     }
```

Note that when logging is disabled logging instance and pointer to that instance are not created.

In order to use the instance logging API in a source file, a compile-time log level must be set using *LOG_LEVEL_SET*.

```
LOG_LEVEL_SET(CONFIG_FOO_LOG_LEVEL);

void foo_init(foo_object *f)
{
```

(continues on next page)

(continued from previous page)

```
    LOG_INST_INF(f->log, "Initialized.");
}
```

In order to use the instance logging API in a header file, a compile-time log level must be set using *LOG_LEVEL_SET*.

```
static inline void foo_init(foo_object *f)
{
    LOG_LEVEL_SET(CONFIG_FOO_LOG_LEVEL);

    LOG_INST_INF(f->log, "Initialized.");
}
```

Controlling the logging

Logging can be controlled using API defined in include/logging/log_ctrl.h. Logger must be initialized before it can be used. Optionally, user can provide function which returns timestamp value. If not provided, *k_cycle_get_32* is used for timestamping. *log_process()* function is used to trigger processing of one log message (if pending). Function returns true if there is more messages pending.

Following snippet shows how logging can be processed in simple forever loop.

```
#include <log_ctrl.h>

void main(void)
{
    log_init();

    while (1) {
            if (log_process() == false) {
                    /* sleep */
            }
    }
}
```

If logs are processed from a thread then it is possible to enable a feature which will wake up processing thread when certain amount of log messages are buffered (see CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD). It is also possible to enable internal logging thread (see CONFIG_LOG_PROCESS_THREAD). In that case, logging thread is initialized and log messages are processed implicitly.

7.15.3 Logging panic

In case of error condition system usually can no longer rely on scheduler or interrupts. In that situation deferred log message processing is not an option. Logger controlling API provides a function for entering into panic mode (*log_panic()*) which should be called in such situation.

When *log_panic()* is called, _panic_ notification is sent to all active backends. Once all backends are notified, all buffered messages are flushed. Since that moment all logs are processed in a blocking way.

7.15.4 Architecture

Logging consists of 3 main parts:

- Frontend

- Core
- Backends

Log message is generated by a source of logging which can be a module or instance of a module.

Default Frontend

Default frontend is engaged when logging API is called in a source of logging (e.g. `LOG_INF`) and is responsible for filtering a message (compile and run time), allocating buffer for the message, creating the message and committing that message. Since logging API can be called in an interrupt, frontend is optimized to log the message as fast as possible.

Log message v1 Each log message consists of one or more fixed size chunks allocated from the pool of fixed size buffers (*Memory Slabs*). Message head chunk contains log entry details like: source ID, timestamp, severity level and the data (string pointer and arguments or raw data). Message contains also a reference counter which indicates how many users still uses this message. It is used to return message to the pool once last user indicates that it can be freed. If more than 3 arguments or 12 bytes of raw data is used in the log then log message is formed from multiple chunks which are linked together. When message body is filled it is put into the list. When log processing is triggered, a message is removed from the list of pending messages. If runtime filtering is disabled, the message is passed to all active backends, otherwise the message is passed to only those backends that have requested messages from that particular source (based on the source ID in the message), and severity level. Once all backends are iterated, the message is considered processed, but the message may still be in use by a backend. Because message is allocated from a pool, it is not mandatory to sequentially free messages. Processing by the backends is asynchronous and memory is freed when last user indicates that message can be freed. It also means that improper backend implementation may lead to pool drought.

Log message v2 Log message v2 contains message descriptor (source, domain and level), timestamp, formatted string details (see *Cbprintf Packaging*) and optional data. Log messages v2 are stored in a continuous block of memory (contrary to v1). Memory is allocated from a circular packet buffer (*Multi Producer Single Consumer Packet Buffer*). It has few consequences:

- Each message is self-contained, continuous block of memory thus it is suited for copying the message (e.g. for offline processing).
- Memory is better utilized because fixed size chunks are not used.
- Messages must be sequentially freed. Backend processing is synchronous. Backend can make a copy for deferred processing.

Log message has following format:

Message Header	2 bits: MPSC packet buffer header
	1 bit: Trace/Log message flag
	3 bits: Domain ID
	3 bits: Level
	10 bits: Cbprintf Package Length
	12 bits: Data length
	1 bit: Reserved
	pointer: Pointer to the source descriptor[1]
	32 or 64 bits: Timestamp[Page 777, 1]
	Optional padding[2]
Cbprintf package (optional)	Header
	Arguments
	Appended strings
Hexdump data (optional)	
Alignment padding (optional)	

Log message allocation It may happen that frontend cannot allocate a message. It happens if system is generating more log messages than it can process in certain time frame. There are two strategies to handle that case:

- No overflow - new log is dropped if space for a message cannot be allocated.
- Overflow - oldest pending messages are freed, until new message can be allocated. Enabled by `CONFIG_LOG_MODE_OVERFLOW`. Note that it degrades performance thus it is recommended to adjust buffer size and amount of enabled logs to limit dropping.

Run-time filtering If run-time filtering is enabled, then for each source of logging a filter structure in RAM is declared. Such filter is using 32 bits divided into ten 3 bit slots. Except *slot 0*, each slot stores current filter for one backend in the system. *Slot 0* (bits 0-2) is used to aggregate maximal filter setting for given source of logging. Aggregate slot determines if log message is created for given entry since it indicates if there is at least one backend expecting that log entry. Backend slots are examined when message is processed by the core to determine if message is accepted by the given backend. Contrary to compile time filtering, binary footprint is increased because logs are compiled in.

In the example below backend 1 is set to receive errors (*slot 1*) and backend 2 up to info level (*slot 2*). Slots 3-9 are not used. Aggregated filter (*slot 0*) is set to info level and up to this level message from that particular source will be buffered.

slot 0	slot 1	slot 2	slot 3	...	slot 9
INF	ERR	INF	OFF	...	OFF

Custom Frontend

Custom frontend is enabled using `CONFIG_LOG_FRONTEND`. Logs are redirected to functions declared in include/logging/log_frontend.h. This may be required in very time-sensitive cases, but most of the logging features cannot be used then, which includes default frontend, core and all backends features.

[1] Depending on the platform and the timestamp size fields may be swapped.
[2] It may be required for cbprintf package alignment

Logging strings

Logging v1 Since log message contains only the value of the argument, when %s is used only the address of a string is stored. Because a string variable argument could be transient, allocated on the stack, or modifiable, logger provides a mechanism and a dedicated buffer pool to hold copies of strings. The buffer size and count is configurable (see CONFIG_LOG_STRDUP_MAX_STRING and CONFIG_LOG_STRDUP_BUF_COUNT).

If a string argument is transient, the user must call *log_strdup()* to duplicate the passed string into a buffer from the pool. See the examples below. If a strdup buffer cannot be allocated, a warning message is logged and an error code returned indicating CONFIG_LOG_STRDUP_BUF_COUNT should be increased. Buffers are freed together with the log message.

```
char local_str[] = "abc";

LOG_INF("logging transient string: %s", log_strdup(local_str));
local_str[0] = '\0'; /* String can be modified, logger will use duplicate."
```

When CONFIG_LOG_DETECT_MISSED_STRDUP is enabled logger will scan each log message and report if string format specifier is found and string address is not in read only memory section or does not belong to memory pool dedicated to string duplicates. It indictes that *log_strdup()* is missing in a call to log a message, such as LOG_INF.

Logging v2 String arguments are handled by *Cbprintf Packaging* thus no special action is required.

Logging backends

Logging backends are registered using *LOG_BACKEND_DEFINE*. The macro creates an instance in the dedicated memory section. Backends can be dynamically enabled (*log_backend_enable()*) and disabled. When *Run-time filtering* is enabled, *log_filter_set()* can be used to dynamically change filtering of a module logs for given backend. Module is identified by source ID and domain ID. Source ID can be retrieved if source name is known by iterating through all registered sources.

Logging supports up to 9 concurrent backends. Log message is passed to the each backend in processing phase. Additionally, backend is notfied when logging enter panic mode with *log_backend_panic()*. On that call backend should switch to synchronous, interrupt-less operation or shut down itself if that is not supported. Occasionally, logging may inform backend about number of dropped messages with *log_backend_dropped()*. Message processing API is version specific.

Logging v1 Logging backend interface contains following functions for processing:

- *log_backend_put()* - backend gets log message in deferred mode.
- *log_backend_put_sync_string()* - backend gets log message with formatted string message in the immediate mode.
- *log_backend_put_sync_hexdump()* - backend gets log message with hexdump message in the immediate mode.

The log message contains a reference counter tracking how many backends are processing the message. On receiving a message backend must claim it by calling *log_msg_get()* on that message which increments a reference counter. Once message is processed, backend puts back the message (*log_msg_put()*) decrementing a reference counter. On last *log_msg_put()*, when reference counter reaches 0, message is returned to the pool. It is up to the backend how message is processed.

Note: The message pool can be starved if a backend does not call *log_msg_put()* when it is done processing a message. The logging core has no means to force messages back to the pool if they're still

marked as in use (with a non-zero reference counter).

```
#include <log_backend.h>

void put(const struct log_backend *const backend,
         struct log_msg *msg)
{
     log_msg_get(msg);

     /* message processing */

     log_msg_put(msg);
}
```

Logging v2 *log_backend_msg2_process()* is used for processing message. It is common for standard and hexdump messages because log message v2 hold string with arguments and data. It is also common for deferred and immediate logging.

Message formatting Logging provides set of function that can be used by the backend to format a message. Helper functions are available in include/logging/log_output.h.

Example message formatted using *log_output_msg_process()* or *log_output_msg2_process()*.

```
[00:00:00.000,274] <info> sample_instance.inst1: logging message
```

Dictionary-based Logging

Dictionary-based logging, instead of human readable texts, outputs the log messages in binary format. This binary format encodes arguments to formatted strings in their native storage formats which can be more compact than their text equivalents. For statically defined strings (including the format strings and any string arguments), references to the ELF file are encoded instead of the whole strings. A dictionary created at build time contains the mappings between these references and the actual strings. This allows the offline parser to obtain the strings from the dictionary to parse the log messages. This binary format allows a more compact representation of log messages in certain scenarios. However, this requires the use of an offline parser and is not as intuitive to use as text-based log messages.

Note that `long double` is not supported by Python's `struct` module. Therefore, log messages with `long double` will not display the correct values.

Configuration Here are kconfig options related to dictionary-based logging:

- `CONFIG_LOG_DICTIONARY_SUPPORT` enables dictionary-based logging support. This should be selected by the backends which require it.
- The UART backend can be used for dictionary-based logging. These are additional config for the UART backend:
  - `CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_HEX` tells the UART backend to output hexadecimal characters for dictionary based logging. This is useful when the log data needs to be captured manually via terminals and consoles.
  - `CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_BIN` tells the UART backend to output binary data.

Usage When dictionary-based logging is enabled via enabling related logging backends, a JSON database file, named `log_dictionary.json`, will be created in the build directory. This database file contains information for the parser to correctly parse the log data. Note that this database file only works with the same build, and cannot be used for any other builds.

To use the log parser:

```
./scripts/logging/dictionary/log_parser.py <build dir>/log_dictionary.json <log data␣
↪file>
```

The parser takes two required arguments, where the first one is the full path to the JSON database file, and the second part is the file containing log data. Add an optional argument `--hex` to the end if the log data file contains hexadecimal characters (e.g. when `CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_HEX=y`). This tells the parser to convert the hexadecimal characters to binary before parsing.

Please refer to logging_dictionary_sample on how to use the log parser.

7.15.5 Limitations and recommendations

Logging v1

The are following limitations:

- Strings as arguments (*%s*) require special treatment (see *Logging strings*).
- Logging double and float variables is not possible because arguments are word size.
- Variables larger than word size cannot be logged.
- Number of arguments in the string is limited to 15.

Logging v2

Solves major limitations of v1. However, in order to get most of the logging capabilities following recommendations shall be followed:

- Enable `CONFIG_LOG_SPEED` to slightly speed up deferred logging at the cost of slight increase in memory footprint.
- Compiler with C11 `_Generic` keyword support is recommended. Logging performance is significantly degraded without it. See *Cbprintf Packaging*.
- When `_Generic` is supported, during compilation it is determined which packaging method shall be used: static or runtime. It is done by searching for any string pointers in the argument list. If string pointer is used with format specifier other than string, e.g. `%p`, it is recommended to cast it to `void *`.

```
LOG_WRN("%s", str);
LOG_WRN("%p", (void *)str);
```

7.15.6 Benchmark

Benchmark numbers from tests/subsys/logging/log_benchmark performed on `qemu_x86`. It is a rough comparison to give general overview. Overall, logging v2 improves in most a the areas with the biggest improvement in logging from userspace. It is at the cost of larger memory footprint for a log message.

Feature	v1	v2	Summary
Kernel logging	7us	7us[3]/11us	No significant change
User logging	86us	13us	**Strongly improved**
kernel logging with overwrite	23us	10us[Page 781, 3]/15us	**Improved**
Logging transient string	16us	42us	**Degraded**
Logging transient string from user	111us	50us	**Improved**
Memory utilization[4]	416	518	Slightly improved
Memory footprint (test)[5]	3.2k	2k	**Improved**
Memory footprint (application)[6]	6.2k	3.5k	**Improved**
Message footprint[7]	15 bytes	47[Page 781, 3]/32 bytes	**Degraded**

Benchmark details

7.15.7 API Reference

Logger API

group `log_api`

Logger API.

Defines

`LOG_ERR(...)`

Writes an ERROR level message to the log.

It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- `...` – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

`LOG_WRN(...)`

Writes a WARNING level message to the log.

It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- `...` – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

`LOG_INF(...)`

Writes an INFO level message to the log.

It's meant to write generic user oriented messages.

Parameters

- `...` – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

[3] `CONFIG_LOG_SPEED` enabled.
[4] Number of log messages with various number of arguments that fits in 2048 bytes dedicated for logging.
[5] Logging subsystem memory footprint in tests/subsys/logging/log_benchmark where filtering and formatting features are not used.
[6] Logging subsystem memory footprint in samples/subsys/logging/logger.
[7] Avarage size of a log message (excluding string) with 2 arguments on `Cortex M3`

LOG_DBG(...)

Writes a DEBUG level message to the log.

It's meant to write developer oriented information.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_PRINTK(...)

Unconditionally print raw log message.

The result is same as if printk was used but it goes through logging infrastructure thus utilizes logging mode, e.g. deferred mode.

Parameters

- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_ERR(_log_inst, ...)

Writes an ERROR level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_WRN(_log_inst, ...)

Writes a WARNING level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_INF(_log_inst, ...)

Writes an INFO level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to write generic user oriented messages.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_INST_DBG(_log_inst, ...)

Writes a DEBUG level message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to write developer oriented information.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- ... – A string optionally containing printk valid conversion specifier, followed by as many values as specifiers.

LOG_HEXDUMP_ERR(_data, _length, _str)

Writes an ERROR level hexdump message to the log.

It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_HEXDUMP_WRN(_data, _length, _str)

Writes a WARNING level message to the log.

It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_HEXDUMP_INF(_data, _length, _str)

Writes an INFO level message to the log.

It's meant to write generic user oriented messages.

Parameters

- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_HEXDUMP_DBG(_data, _length, _str)

Writes a DEBUG level message to the log.

It's meant to write developer oriented information.

Parameters

- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_INST_HEXDUMP_ERR(_log_inst, _data, _length, _str)

Writes an ERROR hexdump message associated with the instance to the log.

Message is associated with specific instance of the module which has independent filtering settings (if runtime filtering is enabled) and message prefix (<module_name>.<instance_name>). It's meant to report severe errors, such as those from which it's not possible to recover.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_INST_HEXDUMP_WRN(_log_inst, _data, _length, _str)

Writes a WARNING level hexdump message associated with the instance to the log.

It's meant to register messages related to unusual situations that are not necessarily errors.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_INST_HEXDUMP_INF(_log_inst, _data, _length, _str)

Writes an INFO level hexdump message associated with the instance to the log.

It's meant to write generic user oriented messages.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_INST_HEXDUMP_DBG(_log_inst, _data, _length, _str)

Writes a DEBUG level hexdump message associated with the instance to the log.

It's meant to write developer oriented information.

Parameters

- _log_inst – Pointer to the log structure associated with the instance.
- _data – Pointer to the data to be logged.
- _length – Length of data (in bytes).
- _str – Persistent, raw string.

LOG_MODULE_REGISTER(...)

Create module-specific state and register the module with Logger.

This macro normally must be used after including <logging/log.h> to complete the initialization of the module.

Module registration can be skipped in two cases:

- The module consists of more than one file, and another file invokes this macro. (*LOG_MODULE_DECLARE()* should be used instead in all of the module's other files.)
- Instance logging is used and there is no need to create module entry. In that case *LOG_LEVEL_SET()* should be used to set log level used within the file.

Macro accepts one or two parameters:

- module name
- optional log level. If not provided then default log level is used in the file.

Example usage:

- *LOG_MODULE_REGISTER(foo, CONFIG_FOO_LOG_LEVEL)*
- *LOG_MODULE_REGISTER(foo)*

See also:

LOG_MODULE_DECLARE

Note: The module's state is defined, and the module is registered, only if LOG_LEVEL for the current source file is non-zero or it is not defined and CONFIG_LOG_DEFAULT_LEVEL is non-zero. In other cases, this macro has no effect.

`LOG_MODULE_DECLARE(...)`

Macro for declaring a log module (not registering it).

Modules which are split up over multiple files must have exactly one file use *LOG_MODULE_REGISTER()* to create module-specific state and register the module with the logger core.

The other files in the module should use this macro instead to declare that same state. (Otherwise, *LOG_INF()* etc. will not be able to refer to module-specific state variables.)

Macro accepts one or two parameters:

- module name
- optional log level. If not provided then default log level is used in the file.

Example usage:

- *LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL)*
- *LOG_MODULE_DECLARE(foo)*

See also:

LOG_MODULE_REGISTER

Note: The module's state is declared only if LOG_LEVEL for the current source file is non-zero or it is not defined and CONFIG_LOG_DEFAULT_LEVEL is non-zero. In other cases, this macro has no effect.

`LOG_LEVEL_SET(level)`

Macro for setting log level in the file or function where instance logging API is used.

Parameters

- `level` – Level used in file or in function.

Functions

static inline char *log_strdup(const char *str)

Logger control

group log_ctrl

Logger control API.

Defines

LOG_CORE_INIT()

LOG_INIT()

LOG_PANIC()

LOG_PROCESS()

Typedefs

typedef log_timestamp_t (*log_timestamp_get_t)(void)

Functions

void log_core_init(void)

Function system initialization of the logger.

Function is called during start up to allow logging before user can explicitly initialize the logger.

void log_init(void)

Function for user initialization of the logger.

void log_thread_set(k_tid_t process_tid)

Function for providing thread which is processing logs.

See CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD.

Note: Function has asserts and has no effect when CONFIG_LOG_PROCESS_THREAD is set.

Parameters

- process_tid – Process thread id. Used to wake up the thread.

int log_set_timestamp_func(*log_timestamp_get_t* timestamp_getter, uint32_t freq)

Function for providing timestamp function.

Parameters

- timestamp_getter – Timestamp function.
- freq – Timestamping frequency.

Returns 0 on success or error.

void log_panic(void)

Switch the logger subsystem to the panic mode.

Returns immediately if the logger is already in the panic mode.

On panic the logger subsystem informs all backends about panic mode. Backends must switch to blocking mode or halt. All pending logs are flushed after switching to panic mode. In panic mode, all log messages must be processed in the context of the call.

bool log_process(bool bypass)

Process one pending log message.

Parameters

- bypass – If true message is released without being processed.

Return values

- true – There is more messages pending to be processed.
- false – No messages pending.

uint32_t log_buffered_cnt(void)

Return number of buffered log messages.

Returns Number of currently buffered log messages.

uint32_t log_src_cnt_get(uint32_t domain_id)

Get number of independent logger sources (modules and instances)

Parameters

- domain_id – Domain ID.

Returns Number of sources.

const char *log_source_name_get(uint32_t domain_id, uint32_t source_id)

Get name of the source (module or instance).

Parameters

- domain_id – Domain ID.
- source_id – Source ID.

Returns Source name or NULL if invalid arguments.

const char *log_domain_name_get(uint32_t domain_id)

Get name of the domain.

Parameters

- domain_id – Domain ID.

Returns Domain name.

uint32_t log_filter_get(struct *log_backend* const *const backend, uint32_t domain_id, int16_t source_id, bool runtime)

Get source filter for the provided backend.

Parameters

- backend – Backend instance.
- domain_id – ID of the domain.
- source_id – Source (module or instance) ID.
- runtime – True for runtime filter or false for compiled in.

Returns Severity level.

uint32_t `log_filter_set`(struct *log_backend* const *const backend, uint32_t domain_id, int16_t source_id, uint32_t level)

Set filter on given source for the provided backend.

Parameters

- `backend` – Backend instance. NULL for all backends.
- `domain_id` – ID of the domain.
- `source_id` – Source (module or instance) ID.
- `level` – Severity level.

Returns Actual level set which may be limited by compiled level. If filter was set for all backends then maximal level that was set is returned.

void `log_backend_enable`(struct *log_backend* const *const backend, void *ctx, uint32_t level)

Enable backend with initial maximum filtering level.

Parameters

- `backend` – Backend instance.
- `ctx` – User context.
- `level` – Severity level.

void `log_backend_disable`(struct *log_backend* const *const backend)

Disable backend.

Parameters

- `backend` – Backend instance.

Log message

group `log_msg`

Log message API.

Defines

`LOG_MAX_NARGS`

Maximum number of arguments in the standard log entry.

It is limited by 4 bit nargs field in the log message.

`LOG_MSG_NARGS_SINGLE_CHUNK`

Number of arguments in the log entry which fits in one chunk.

`LOG_MSG_NARGS_HEAD_CHUNK`

Number of arguments in the head of extended standard log message..

`LOG_MSG_HEXDUMP_BYTES_SINGLE_CHUNK`

Maximal amount of bytes in the hexdump entry which fits in one chunk.

`LOG_MSG_HEXDUMP_BYTES_HEAD_CHUNK`

Number of bytes in the first chunk of hexdump message if message consists of more than one chunk.

HEXDUMP_BYTES_CONT_MSG

Number of bytes that can be stored in chunks following head chunk in hexdump log message.

ARGS_CONT_MSG

LOG_MSG_TYPE_STD

Flag indicating standard log message.

LOG_MSG_TYPE_HEXDUMP

Flag indicating hexdump log message.

COMMON_PARAM_HDR()

Common part of log message header.

LOG_MSG_HEXDUMP_LENGTH_BITS

Number of bits used for storing length of hexdump log message.

LOG_MSG_HEXDUMP_MAX_LENGTH

Maximum length of log hexdump message.

Typedefs

typedef unsigned long log_arg_t

Log argument type.

Should preferably be equivalent to a native word size.

Functions

void log_msg_pool_init(void)

Function for initialization of the log message pool.

void log_msg_get(struct *log_msg* *msg)

Function for indicating that message is in use.

Message can be used (read) by multiple users. Internal reference counter is atomically increased. See *log_msg_put*.

Parameters

- msg – Message.

void log_msg_put(struct *log_msg* *msg)

Function for indicating that message is no longer in use.

Internal reference counter is atomically decreased. If reference counter equals 0 message is freed.

Parameters

- msg – Message.

static inline uint32_t log_msg_domain_id_get(struct *log_msg* *msg)

Get domain ID of the message.

Parameters

- `msg` – Message

Returns Domain ID.

static inline uint32_t `log_msg_source_id_get`(struct *log_msg* *msg)

Get source ID (module or instance) of the message.

Parameters

- `msg` – Message

Returns Source ID.

static inline uint32_t `log_msg_level_get`(struct *log_msg* *msg)

Get severity level of the message.

Parameters

- `msg` – Message

Returns Severity message.

static inline uint32_t `log_msg_timestamp_get`(struct *log_msg* *msg)

Get timestamp of the message.

Parameters

- `msg` – Message

Returns Timestamp value.

static inline bool `log_msg_is_std`(struct *log_msg* *msg)

Check if message is of standard type.

Parameters

- `msg` – Message

Return values

- `true` – Standard message.
- `false` – Hexdump message.

uint32_t `log_msg_nargs_get`(struct *log_msg* *msg)

Returns number of arguments in standard log message.

Parameters

- `msg` – Standard log message.

Returns Number of arguments.

log_arg_t `log_msg_arg_get`(struct *log_msg* *msg, uint32_t arg_idx)

Gets argument from standard log message.

Parameters

- `msg` – Standard log message.
- `arg_idx` – Argument index.

Returns Argument value or 0 if arg_idx exceeds number of arguments in the message.

const char *`log_msg_str_get`(struct *log_msg* *msg)

Gets pointer to the unformatted string from standard log message.

Parameters

- `msg` – Standard log message.

Returns Pointer to the string.

struct *log_msg* *log_msg_hexdump_create(const char *str, const uint8_t *data, uint32_t length)

Allocates chunks for hexdump message and copies the data.

Function resets header and sets following fields:

- message type
- length

Note: Allocation and partial filling is combined for performance reasons.

Parameters

- `str` – String.
- `data` – Data.
- `length` – Data length.

Returns Pointer to allocated head of the message or NULL

void log_msg_hexdump_data_put(struct *log_msg* *msg, uint8_t *data, size_t *length, size_t offset)

Put data into hexdump log message.

Parameters

- `msg` – **[in]** Message.
- `data` – **[in]** Data to be copied.
- `length` – **[inout]** Input: requested amount. Output: actual amount.
- `offset` – **[in]** Offset.

void log_msg_hexdump_data_get(struct *log_msg* *msg, uint8_t *data, size_t *length, size_t offset)

Get data from hexdump log message.

Parameters

- `msg` – **[in]** Message.
- `data` – **[in]** Buffer for data.
- `length` – **[inout]** Input: requested amount. Output: actual amount.
- `offset` – **[in]** Offset.

union *log_msg_chunk* *log_msg_no_space_handle(void)

union *log_msg_chunk* *log_msg_chunk_alloc(void)

Allocate single chunk from the pool.

Returns Pointer to the allocated chunk or NULL if failed to allocate.

static inline struct *log_msg* *log_msg_create_0(const char *str)

Create standard log message with no arguments.

Function resets header and sets following fields:

- message type
- string pointer

Returns Pointer to allocated head of the message or NULL.

static inline struct *log_msg* *log_msg_create_1(const char *str, *log_arg_t* arg1)

Create standard log message with one argument.

Function resets header and sets following fields:

- message type
- string pointer
- number of arguments
- argument

Parameters

- str – String.
- arg1 – Argument.

Returns Pointer to allocated head of the message or NULL.

static inline struct *log_msg* *log_msg_create_2(const char *str, *log_arg_t* arg1, *log_arg_t* arg2)

Create standard log message with two arguments.

Function resets header and sets following fields:

- message type
- string pointer
- number of arguments
- arguments

Parameters

- str – String.
- arg1 – Argument 1.
- arg2 – Argument 2.

Returns Pointer to allocated head of the message or NULL.

static inline struct *log_msg* *log_msg_create_3(const char *str, *log_arg_t* arg1, *log_arg_t* arg2, *log_arg_t* arg3)

Create standard log message with three arguments.

Function resets header and sets following fields:

- message type
- string pointer
- number of arguments
- arguments

Parameters

- str – String.
- arg1 – Argument 1.
- arg2 – Argument 2.
- arg3 – Argument 3.

Returns Pointer to allocated head of the message or NULL.

struct *log_msg* *log_msg_create_n(const char *str, *log_arg_t* *args, uint32_t nargs)

Create standard log message with variable number of arguments.

Function resets header and sets following fields:

- message type
- string pointer
- number of arguments
- arguments

Parameters

- `str` – String.
- `args` – Array with arguments.
- `nargs` – Number of arguments.

Returns Pointer to allocated head of the message or NULL.

uint32_t log_msg_mem_get_free(void)

Get number of free blocks from the log mem pool.

uint32_t log_msg_mem_get_used(void)

Get number of used blocks from the log mem pool.

uint32_t log_msg_mem_get_max_used(void)

Get max used blocks from the log mem pool.

struct log_msg_ids

#include <log_msg.h> Part of log message header identifying source and level.

Public Members

uint16_t level

Severity.

uint16_t domain_id

Originating domain.

uint16_t source_id

Source ID.

struct log_msg_generic_hdr

#include <log_msg.h> Part of log message header common to standard and hexdump log message.

struct log_msg_std_hdr

#include <log_msg.h> Part of log message header specific to standard log message.

struct log_msg_hexdump_hdr

#include <log_msg.h> Part of log message header specific to hexdump log message.

struct log_msg_hdr

#include <log_msg.h> Log message header structure

Public Members

atomic_t `ref_cnt`
Reference counter for tracking message users.

struct *log_msg_ids* `ids`
Identification part of the message.

uint32_t `timestamp`
Timestamp.

union `log_msg_hdr_params`
#include <log_msg.h>

Public Members

struct *log_msg_generic_hdr* `generic`

struct *log_msg_std_hdr* `std`

struct *log_msg_hexdump_hdr* `hexdump`

uint16_t `raw`

union `log_msg_head_data`
#include <log_msg.h> Data part of log message.

Public Members

log_arg_t `args`[3U]

uint8_t `bytes`[(3U * sizeof(*log_arg_t*))]

struct `log_msg_ext_head_data`
#include <log_msg.h> Data part of extended log message.

union `log_msg_ext_head_data_data`
#include <log_msg.h>

Public Members

log_arg_t `args`[(3U - (sizeof(void*) / sizeof(*log_arg_t*)))]

uint8_t `bytes`[((3U * sizeof(*log_arg_t*)) - sizeof(void*))]

struct log_msg

#include <log_msg.h> Log message structure.

Public Members

struct *log_msg* *next

Used by logger core list.

struct *log_msg_hdr* hdr

Message header.

union *log_msg*.*log_msg_data* payload

Message data.

union log_msg_data

#include <log_msg.h>

Public Members

union *log_msg_head_data* single

struct *log_msg_ext_head_data* ext

struct log_msg_cont

#include <log_msg.h> Chunks following message head when message is extended.

Public Members

struct *log_msg_cont* *next

Pointer to the next chunk.

union log_msg_cont_data

#include <log_msg.h>

Public Members

log_arg_t args[((sizeof(struct *log_msg*) - sizeof(void*)) / sizeof(*log_arg_t*))]

uint8_t bytes[(sizeof(struct *log_msg*) - sizeof(void*))]

union log_msg_chunk

#include <log_msg.h> Log message.

Public Members

struct *log_msg* `head`

struct *log_msg_cont* `cont`

Logger backend interface

group `log_backend`

Logger backend interface.

Defines

`LOG_BACKEND_DEFINE(_name, _api, _autostart, ...)`

Macro for creating a logger backend instance.

Parameters

- `_name` – Name of the backend instance.
- `_api` – Logger backend API.
- `_autostart` – If true backend is initialized and activated together with the logger subsystem.
- `...` – Optional context.

Functions

static inline void `log_backend_put`(const struct *log_backend* *const backend, struct *log_msg* *msg)

Put message with log entry to the backend.

Parameters

- `backend` – **[in]** Pointer to the backend instance.
- `msg` – **[in]** Pointer to message with log entry.

static inline void `log_backend_msg2_process`(const struct *log_backend* *const backend, union log_msg2_generic *msg)

static inline void `log_backend_put_sync_string`(const struct *log_backend* *const backend, struct *log_msg_ids* src_level, uint32_t timestamp, const char *fmt, va_list ap)

Synchronously process log message.

Parameters

- `backend` – **[in]** Pointer to the backend instance.
- `src_level` – **[in]** Message details.
- `timestamp` – **[in]** Timestamp.
- `fmt` – **[in]** Log string.
- `ap` – **[in]** Log string arguments.

static inline void `log_backend_put_sync_hexdump`(const struct *log_backend* *const backend, struct *log_msg_ids* src_level, uint32_t timestamp, const char *metadata, const uint8_t *data, uint32_t len)

Synchronously process log hexdump_message.

Parameters

- `backend` – **[in]** Pointer to the backend instance.
- `src_level` – **[in]** Message details.
- `timestamp` – **[in]** Timestamp.
- `metadata` – **[in]** Raw string associated with the data.
- `data` – **[in]** Data.
- `len` – **[in]** Data length.

static inline void `log_backend_dropped`(const struct *log_backend* *const backend, uint32_t cnt)

Notify backend about dropped log messages.

Function is optional.

Parameters

- `backend` – **[in]** Pointer to the backend instance.
- `cnt` – **[in]** Number of dropped logs since last notification.

static inline void `log_backend_panic`(const struct *log_backend* *const backend)

Reconfigure backend to panic mode.

Parameters

- `backend` – **[in]** Pointer to the backend instance.

static inline void `log_backend_id_set`(const struct *log_backend* *const backend, uint8_t id)

Set backend id.

Note: It is used internally by the logger.

Parameters

- `backend` – Pointer to the backend instance.
- `id` – ID.

static inline uint8_t `log_backend_id_get`(const struct *log_backend* *const backend)

Get backend id.

Note: It is used internally by the logger.

Parameters

- `backend` – **[in]** Pointer to the backend instance.

Returns Id.

static inline const struct *log_backend* *log_backend_get(uint32_t idx)

Get backend.

Parameters

- idx – **[in]** Pointer to the backend instance.

Returns Pointer to the backend instance.

static inline int log_backend_count_get(void)

Get number of backends.

Returns Number of backends.

static inline void log_backend_activate(const struct *log_backend* *const backend, void *ctx)

Activate backend.

Parameters

- backend – **[in]** Pointer to the backend instance.
- ctx – **[in]** User context.

static inline void log_backend_deactivate(const struct *log_backend* *const backend)

Deactivate backend.

Parameters

- backend – **[in]** Pointer to the backend instance.

static inline bool log_backend_is_active(const struct *log_backend* *const backend)

Check state of the backend.

Parameters

- backend – **[in]** Pointer to the backend instance.

Returns True if backend is active, false otherwise.

struct log_backend_api

#include <log_backend.h> Logger backend API.

struct log_backend_control_block

#include <log_backend.h> Logger backend control block.

struct log_backend

#include <log_backend.h> Logger backend structure.

Logger output formatting

group log_output

Log output API.

Defines

LOG_OUTPUT_FLAG_COLORS

Flag forcing ANSI escape code colors, red (errors), yellow (warnings).

LOG_OUTPUT_FLAG_TIMESTAMP

Flag forcing timestamp.

LOG_OUTPUT_FLAG_FORMAT_TIMESTAMP

Flag forcing timestamp formatting.

LOG_OUTPUT_FLAG_LEVEL

Flag forcing severity level prefix.

LOG_OUTPUT_FLAG_CRLF_NONE

Flag preventing the logger from adding CR and LF characters.

LOG_OUTPUT_FLAG_CRLF_LFONLY

Flag forcing a single LF character for line breaks.

LOG_OUTPUT_FLAG_FORMAT_SYSLOG

Flag forcing syslog format specified in RFC 5424.

LOG_OUTPUT_FLAG_FORMAT_SYST

Flag forcing syslog format specified in mipi sys-t.

LOG_OUTPUT_DEFINE(_name, _func, _buf, _size)

Create *log_output* instance.

Parameters

- _name – Instance name.
- _func – Function for processing output data.
- _buf – Pointer to the output buffer.
- _size – Size of the output buffer.

Typedefs

typedef int (*log_output_func_t)(uint8_t *buf, size_t size, void *ctx)

Prototype of the function processing output data.

Note: If the log output function cannot process all of the data, it is its responsibility to mark them as dropped or discarded by returning the corresponding number of bytes dropped or discarded to the caller.

Param buf The buffer data.

Param size The buffer size.

Param ctx User context.

Return Number of bytes processed, dropped or discarded.

Functions

void `log_output_msg_process`(const struct *log_output* *output, struct *log_msg* *msg, uint32_t flags)

Process log messages to readable strings.

Function is using provided context with the buffer and output function to process formatted string and output the data.

Parameters

- `output` – Pointer to the log output instance.
- `msg` – Log message.
- `flags` – Optional flags.

void `log_output_msg2_process`(const struct *log_output* *log_output, struct log_msg2 *msg, uint32_t flags)

Process log messages v2 to readable strings.

Function is using provided context with the buffer and output function to process formatted string and output the data.

Parameters

- `log_output` – Pointer to the log output instance.
- `msg` – Log message.
- `flags` – Optional flags.

void `log_output_string`(const struct *log_output* *output, struct *log_msg_ids* src_level, uint32_t timestamp, const char *fmt, va_list ap, uint32_t flags)

Process log string.

Function is formatting provided string adding optional prefixes and postfixes.

Parameters

- `output` – Pointer to *log_output* instance.
- `src_level` – Log source and level structure.
- `timestamp` – Timestamp.
- `fmt` – String.
- `ap` – String arguments.
- `flags` – Optional flags.

void `log_output_hexdump`(const struct *log_output* *output, struct *log_msg_ids* src_level, uint32_t timestamp, const char *metadata, const uint8_t *data, uint32_t length, uint32_t flags)

Process log hexdump.

Function is formatting provided hexdump adding optional prefixes and postfixes.

Parameters

- `output` – Pointer to *log_output* instance.
- `src_level` – Log source and level structure.
- `timestamp` – Timestamp.
- `metadata` – String.
- `data` – Data.

- `length` – Data length.
- `flags` – Optional flags.

void `log_output_dropped_process`(const struct *log_output* *output, uint32_t cnt)

Process dropped messages indication.

Function prints error message indicating lost log messages.

Parameters

- `output` – Pointer to the log output instance.
- `cnt` – Number of dropped messages.

void `log_output_flush`(const struct *log_output* *output)

Flush output buffer.

Parameters

- `output` – Pointer to the log output instance.

static inline void `log_output_ctx_set`(const struct *log_output* *output, void *ctx)

Function for setting user context passed to the output function.

Parameters

- `output` – Pointer to the log output instance.
- `ctx` – User context.

static inline void `log_output_hostname_set`(const struct *log_output* *output, const char *hostname)

Function for setting hostname of this device.

Parameters

- `output` – Pointer to the log output instance.
- `hostname` – Hostname of this device

void `log_output_timestamp_freq_set`(uint32_t freq)

Set timestamp frequency.

Parameters

- `freq` – Frequency in Hz.

uint64_t `log_output_timestamp_to_us`(uint32_t timestamp)

Convert timestamp of the message to us.

Parameters

- `timestamp` – Message timestamp

Returns Timestamp value in us.

struct `log_output_control_block`

#include <log_output.h>

struct `log_output`

#include <log_output.h> Log_output instance structure.

7.16 Memory Management

The following contains various topics regarding memory management.

7.16.1 Demand Paging

Demand paging provides a mechanism where data is only brought into physical memory as required by current execution context. The physical memory is conceptually divided in page-sized page frames as regions to hold data.

- When the processor tries to access data and the data page exists in one of the page frames, the execution continues without any interruptions.
- When the processor tries to access the data page that does not exist in any page frames, a page fault occurs. The paging code then brings in the corresponding data page from backing store into physical memory if there is a free page frame. If there is no more free page frames, the eviction algorithm is invoked to select a data page to be paged out, thus freeing up a page frame for new data to be paged in. If this data page has been modified after it is first paged in, the data will be written back into the backing store. If no modifications is done or after written back into backing store, the data page is now considered paged out and the corresponding page frame is now free. The paging code then invokes the backing store to page in the data page corresponding to the location of the requested data. The backing store copies that data page into the free page frame. Now the data page is in physical memory and execution can continue.

There are functions where paging in and out can be invoked manually using *k_mem_page_in()* and *k_mem_page_out()*. *k_mem_page_in()* can be used to page in data pages in anticipation that they are required in the near future. This is used to minimize number of page faults as these data pages are already in physical memory, and thus minimizing latency. *k_mem_page_out()* can be used to page out data pages where they are not going to be accessed for a considerable amount of time. This frees up page frames so that the next page in can be executed faster as the paging code does not need to invoke the eviction algorithm.

Terminology

Data Page A data page is a page-sized region of data. It may exist in a page frame, or be paged out to some backing store. Its location can always be looked up in the CPU's page tables (or equivalent) by virtual address. The data type will always be `void *` or in some cases `uint8_t *` when doing pointer arithmetic.

Page Frame A page frame is a page-sized physical memory region in RAM. It is a container where a data page may be placed. It is always referred to by physical address. Zephyr has a convention of using `uintptr_t` for physical addresses. For every page frame, a `struct z_page_frame` is instantiated to store metadata. Flags for each page frame:

- `Z_PAGE_FRAME_PINNED` indicates a page frame is pinned in memory and should never be paged out.
- `Z_PAGE_FRAME_RESERVED` indicates a physical page reserved by hardware and should not be used at all.
- `Z_PAGE_FRAME_MAPPED` is set when a physical page is mapped to virtual memory address.
- `Z_PAGE_FRAME_BUSY` indicates a page frame is currently involved in a page-in/out operation.
- `Z_PAGE_FRAME_BACKED` indicates a page frame has a clean copy in the backing store.

Z_SCRATCH_PAGE The virtual address of a special page provided to the backing store to: * Copy a data page from `Z_SCRATCH_PAGE` to the specified location; or, * Copy a data page from the provided location to `Z_SCRATCH_PAGE`. This is used as an intermediate page for page in/out operations. This scratch needs to be mapped read/write for backing store code to access. However the data page itself may only be mapped as read-only in virtual address space. If this page is provided as-is to backing store, the data page must be re-mapped as read/write which has security implications as the data page is no longer read-only to other parts of the application.

Paging Statistics

Paging statistics can be obtained via various function calls when CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM_NUM_BINS is enabled:

- Overall statistics via *k_mem_paging_stats_get()*
- Per-thread statistics via *k_mem_paging_thread_stats_get()* if CONFIG_DEMAND_PAGING_THREAD_STATS is enabled
- Execution time histogram can be obtained when CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM is enabled, and CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM_NUM_BINS is defined. Note that the timing is highly dependent on the architecture, SoC or board. It is highly recommended that k_mem_paging_eviction_histogram_bounds[] and k_mem_paging_backing_store_histogram_bounds[] be defined for a particular application.
  - Execution time histogram of eviction algorithm via *k_mem_paging_histogram_eviction_get()*
  - Execution time histogram of backing store doing page-in via *k_mem_paging_histogram_backing_store_page_in_get()*
  - Execution time histogram of backing store doing page-out via *k_mem_paging_histogram_backing_store_page_out_get()*

Eviction Algorithm

The eviction algorithm is used to determine which data page and its corresponding page frame can be paged out to free up a page frame for the next page in operation. There are two functions which are called from the kernel paging code:

- *k_mem_paging_eviction_init()* is called to initialize the eviction algorithm. This is called at POST_KERNEL.
- *k_mem_paging_eviction_select()* is called to select a data page to evict. A function argument dirty is written to signal the caller whether the selected data page has been modified since it is first paged in. If the dirty bit is returned as set, the paging code signals to the backing store to write the data page back into storage (thus updating its content). The function returns a pointer to the page frame corresponding to the selected data page.

Currently, a NRU (Not-Recently-Used) eviction algorithm has been implemented as a sample. This is a very simple algorithm which ranks each data page on whether they have been accessed and modified. The selection is based on this ranking.

To implement a new eviction algorithm, the two functions mentioned above must be implemented.

Backing Store

Backing store is responsible for paging in/out data page between their corresponding page frames and storage. These are the functions which must be implemented:

- *k_mem_paging_backing_store_init()* is called to initialized the backing store at POST_KERNEL.
- *k_mem_paging_backing_store_location_get()* is called to reserve a backing store location so a data page can be paged out. This location token is passed to *k_mem_paging_backing_store_page_out()* to perform actual page out operation.
- *k_mem_paging_backing_store_location_free()* is called to free a backing store location (the location token) which can then be used for subsequent page out operation.
- *k_mem_paging_backing_store_page_in()* copies a data page from the backing store location associated with the provided location token to the page pointed by Z_SCRATCH_PAGE.
- *k_mem_paging_backing_store_page_out()* copies a data page from Z_SCRATCH_PAGE to the backing store location associated with the provided location token.

- *k_mem_paging_backing_store_page_finalize()* is invoked after *k_mem_paging_backing_store_page_in()* so that the page frame struct may be updated for internal accounting. This can be a no-op.

To implement a new backing store, the functions mentioned above must be implemented. *k_mem_paging_backing_store_page_finalize()* can be an empty function if so desired.

API Reference

group mem-demand-paging

Functions

int k_mem_page_out(void *addr, size_t size)

Evict a page-aligned virtual memory region to the backing store

Useful if it is known that a memory region will not be used for some time. All the data pages within the specified region will be evicted to the backing store if they weren't already, with their associated page frames marked as available for mappings or page-ins.

None of the associated page frames mapped to the provided region should be pinned.

Note that there are no guarantees how long these pages will be evicted, they could take page faults immediately.

If CONFIG_DEMAND_PAGING_ALLOW_IRQ is enabled, this function may not be called by ISRs as the backing store may be in-use.

Parameters

- addr – Base page-aligned virtual address
- size – Page-aligned data region size

Return values

- 0 – Success
- -ENOMEM – Insufficient space in backing store to satisfy request. The region may be partially paged out.

void k_mem_page_in(void *addr, size_t size)

Load a virtual data region into memory

After the function completes, all the page frames associated with this function will be paged in. However, they are not guaranteed to stay there. This is useful if the region is known to be used soon.

If CONFIG_DEMAND_PAGING_ALLOW_IRQ is enabled, this function may not be called by ISRs as the backing store may be in-use.

Parameters

- addr – Base page-aligned virtual address
- size – Page-aligned data region size

void k_mem_pin(void *addr, size_t size)

Pin an aligned virtual data region, paging in as necessary

After the function completes, all the page frames associated with this region will be resident in memory and pinned such that they stay that way. This is a stronger version of z_mem_page_in().

If CONFIG_DEMAND_PAGING_ALLOW_IRQ is enabled, this function may not be called by ISRs as the backing store may be in-use.

Parameters

- `addr` – Base page-aligned virtual address
- `size` – Page-aligned data region size

void `k_mem_unpin`(void *addr, size_t size)

Un-pin an aligned virtual data region

After the function completes, all the page frames associated with this region will be no longer marked as pinned. This does not evict the region, follow this with z_mem_page_out() if you need that.

Parameters

- `addr` – Base page-aligned virtual address
- `size` – Page-aligned data region size

void `k_mem_paging_stats_get`(struct k_mem_paging_stats_t *stats)

Get the paging statistics since system startup

This populates the paging statistics struct being passed in as argument.

Parameters

- `stats` – **[inout]** Paging statistics struct to be filled.

void `k_mem_paging_thread_stats_get`(struct *k_thread* *thread, struct k_mem_paging_stats_t *stats)

Get the paging statistics since system startup for a thread

This populates the paging statistics struct being passed in as argument for a particular thread.

Parameters

- `thread` – **[in]** Thread
- `stats` – **[inout]** Paging statistics struct to be filled.

void `k_mem_paging_histogram_eviction_get`(struct k_mem_paging_histogram_t *hist)

Get the eviction timing histogram

This populates the timing histogram struct being passed in as argument.

Parameters

- `hist` – **[inout]** Timing histogram struct to be filled.

void `k_mem_paging_histogram_backing_store_page_in_get`(struct k_mem_paging_histogram_t *hist)

Get the backing store page-in timing histogram

This populates the timing histogram struct being passed in as argument.

Parameters

- `hist` – **[inout]** Timing histogram struct to be filled.

void `k_mem_paging_histogram_backing_store_page_out_get`(struct k_mem_paging_histogram_t *hist)

Get the backing store page-out timing histogram

This populates the timing histogram struct being passed in as argument.

Parameters

- `hist` – **[inout]** Timing histogram struct to be filled.

Eviction Algorithm APIs

group `mem-demand-paging-eviction`

Eviction algorithm APIs

Functions

struct z_page_frame *`k_mem_paging_eviction_select`(bool *dirty)

Select a page frame for eviction

The kernel will invoke this to choose a page frame to evict if there are no free page frames.

This function will never be called before the initial *k_mem_paging_eviction_init()*.

This function is invoked with interrupts locked.

Parameters

- `dirty` – **[out]** Whether the page to evict is dirty

Returns The page frame to evict

void `k_mem_paging_eviction_init`(void)

Initialization function

Called at POST_KERNEL to perform any necessary initialization tasks for the eviction algorithm. *k_mem_paging_eviction_select()* is guaranteed to never be called until this has returned, and this will only be called once.

Backing Store APIs

group `mem-demand-paging-backing-store`

Backing store APIs

Functions

int `k_mem_paging_backing_store_location_get`(struct z_page_frame *pf, uintptr_t *location, bool page_fault)

Reserve or fetch a storage location for a data page loaded into a page frame

The returned location token must be unique to the mapped virtual address. This location will be used in the backing store to page out data page contents for later retrieval. The location value must be page-aligned.

This function may be called multiple times on the same data page. If its page frame has its Z_PAGE_FRAME_BACKED bit set, it is expected to return the previous backing store location for the data page containing a cached clean copy. This clean copy may be updated on page-out, or used to discard clean pages without needing to write out their contents.

If the backing store is full, some other backing store location which caches a loaded data page may be selected, in which case its associated page frame will have the Z_PAGE_FRAME_BACKED bit cleared (as it is no longer cached).

pf->addr will indicate the virtual address the page is currently mapped to. Large, sparse backing stores which can contain the entire address space may simply generate location tokens purely as a function of pf->addr with no other management necessary.

This function distinguishes whether it was called on behalf of a page fault. A free backing store location must always be reserved in order for page faults to succeed. If the page_fault parameter is not set, this function should return -ENOMEM even if one location is available.

This function is invoked with interrupts locked.

Parameters

- `pf` – Virtual address to obtain a storage location
- `location` – **[out]** storage location token
- `page_fault` – Whether this request was for a page fault

Returns 0 Success

Returns -ENOMEM Backing store is full

void `k_mem_paging_backing_store_location_free`(uintptr_t location)

Free a backing store location

Any stored data may be discarded, and the location token associated with this address may be re-used for some other data page.

This function is invoked with interrupts locked.

Parameters

- `location` – Location token to free

void `k_mem_paging_backing_store_page_out`(uintptr_t location)

Copy a data page from Z_SCRATCH_PAGE to the specified location

Immediately before this is called, Z_SCRATCH_PAGE will be mapped read-write to the intended source page frame for the calling context.

Calls to this and *k_mem_paging_backing_store_page_in()* will always be serialized, but interrupts may be enabled.

Parameters

- `location` – Location token for the data page, for later retrieval

void `k_mem_paging_backing_store_page_in`(uintptr_t location)

Copy a data page from the provided location to Z_SCRATCH_PAGE.

Immediately before this is called, Z_SCRATCH_PAGE will be mapped read-write to the intended destination page frame for the calling context.

Calls to this and *k_mem_paging_backing_store_page_out()* will always be serialized, but interrupts may be enabled.

Parameters

- `location` – Location token for the data page

void `k_mem_paging_backing_store_page_finalize`(struct z_page_frame *pf, uintptr_t location)

Update internal accounting after a page-in

This is invoked after *k_mem_paging_backing_store_page_in()* and interrupts have been* re-locked, making it safe to access the z_page_frame data. The location value will be the same passed to *k_mem_paging_backing_store_page_in()*.

The primary use-case for this is to update custom fields for the backing store in the page frame, to reflect where the data should be evicted to if it is paged out again. This may be a no-op in some implementations.

If the backing store caches paged-in data pages, this is the appropriate time to set the Z_PAGE_FRAME_BACKED bit. The kernel only skips paging out clean data pages if they

are noted as clean in the page tables and the Z_PAGE_FRAME_BACKED bit is set in their associated page frame.

Parameters

- `pf` – Page frame that was loaded in
- `location` – Location of where the loaded data page was retrieved

void `k_mem_paging_backing_store_init(void)`

Backing store initialization function.

The implementation may expect to receive page in/out calls as soon as this returns, but not before that. Called at POST_KERNEL.

This function is expected to do two things:

- Initialize any internal data structures and accounting for the backing store.
- If the backing store already contains all or some loaded kernel data pages at boot time, Z_PAGE_FRAME_BACKED should be appropriately set for their associated page frames, and any internal accounting set up appropriately.

7.17 Miscellaneous APIs

7.17.1 Checksum APIs

CRC

group `crc`

Functions

uint16_t `crc16(const uint8_t *src, size_t len, uint16_t polynomial, uint16_t initial_value, bool pad)`

Generic function for computing CRC 16.

Compute CRC 16 by passing in the address of the input, the input length and polynomial used in addition to the initial value.

Parameters

- `src` – Input bytes for the computation
- `len` – Length of the input in bytes
- `polynomial` – The polynomial to use omitting the leading x^16 coefficient
- `initial_value` – Initial value for the CRC computation
- `pad` – Adds padding with zeros at the end of input bytes

Returns The computed CRC16 value

uint8_t `crc8(const uint8_t *src, size_t len, uint8_t polynomial, uint8_t initial_value, bool reversed)`

Generic function for computing CRC 8.

Compute CRC 8 by passing in the address of the input, the input length and polynomial used in addition to the initial value.

Parameters

- `src` – Input bytes for the computation
- `len` – Length of the input in bytes
- `polynomial` – The polynomial to use omitting the leading x^8 coefficient
- `initial_value` – Initial value for the CRC computation
- `reversed` – Should we use reflected/reversed values or not

Returns The computed CRC8 value

uint16_t `crc16_ccitt`(uint16_t seed, const uint8_t *src, size_t len)

Compute the CRC-16/CCITT checksum of a buffer.

See ITU-T Recommendation V.41 (November 1988). Uses 0x1021 as the polynomial, reflects the input, and reflects the output.

To calculate the CRC across non-contiguous blocks use the return value from block N-1 as the seed for block N.

For CRC-16/CCITT, use 0 as the initial seed. Other checksums in the same family can be calculated by changing the seed and/or XORing the final value. Examples include:

- X-25 (used in PPP): seed=0xffff, xor=0xffff, residual=0xf0b8

Note: API changed in Zephyr 1.11.

Parameters

- `seed` – Value to seed the CRC with
- `src` – Input bytes for the computation
- `len` – Length of the input in bytes

Returns The computed CRC16 value

uint16_t `crc16_itu_t`(uint16_t seed, const uint8_t *src, size_t len)

Compute the CRC-16/XMODEM checksum of a buffer.

The MSB first version of ITU-T Recommendation V.41 (November 1988). Uses 0x1021 as the polynomial with no reflection.

To calculate the CRC across non-contiguous blocks use the return value from block N-1 as the seed for block N.

For CRC-16/XMODEM, use 0 as the initial seed. Other checksums in the same family can be calculated by changing the seed and/or XORing the final value. Examples include:

- CCIITT-FALSE: seed=0xffff
- GSM: seed=0, xorout=0xffff, residue=0x1d0f

Parameters

- `seed` – Value to seed the CRC with
- `src` – Input bytes for the computation
- `len` – Length of the input in bytes

Returns The computed CRC16 value

static inline uint16_t crc16_ansi(const uint8_t *src, size_t len)

Compute ANSI variant of CRC 16.

ANSI variant of CRC 16 is using 0x8005 as its polynomial with the initial value set to 0xffff.

Parameters

- src – Input bytes for the computation
- len – Length of the input in bytes

Returns The computed CRC16 value

uint32_t crc32_ieee(const uint8_t *data, size_t len)

Generate IEEE conform CRC32 checksum.

Parameters

- *data – Pointer to data on which the CRC should be calculated.
- len – Data length.

Returns CRC32 value.

uint32_t crc32_ieee_update(uint32_t crc, const uint8_t *data, size_t len)

Update an IEEE conforming CRC32 checksum.

Parameters

- crc – CRC32 checksum that needs to be updated.
- *data – Pointer to data on which the CRC should be calculated.
- len – Data length.

Returns CRC32 value.

uint32_t crc32_c(uint32_t crc, const uint8_t *data, size_t len, bool first_pkt, bool last_pkt)

Calculate CRC32C (Castagnoli) checksum.

Parameters

- crc – CRC32C checksum that needs to be updated.
- *data – Pointer to data on which the CRC should be calculated.
- len – Data length.
- first_pkt – Whether this is the first packet in the stream.
- last_pkt – Whether this is the last packet in the stream.

Returns CRC32 value.

uint8_t crc8_ccitt(uint8_t initial_value, const void *buf, size_t len)

Compute CCITT variant of CRC 8.

Normal CCITT variant of CRC 8 is using 0x07.

Parameters

- initial_value – Initial value for the CRC computation
- buf – Input bytes for the computation
- len – Length of the input in bytes

Returns The computed CRC8 value

uint8_t crc7_be(uint8_t seed, const uint8_t *src, size_t len)

Compute the CRC-7 checksum of a buffer.

See JESD84-A441. Used by the MMC protocol. Uses 0x09 as the polynomial with no reflection. The CRC is left justified, so bit 7 of the result is bit 6 of the CRC.

Parameters

- seed – Value to seed the CRC with
- src – Input bytes for the computation
- len – Length of the input in bytes

Returns The computed CRC7 value

7.17.2 Structured Data APIs

JSON

group json

Defines

JSON_OBJ_DESCR_PRIM(struct_, field_name_, type_)

Helper macro to declare a descriptor for supported primitive values.

Here's an example of use:

```
struct foo {
    int some_int;
};

struct json_obj_descr foo[] = {
    JSON_OBJ_DESCR_PRIM(struct foo, some_int, JSON_TOK_NUMBER),
};
```

Parameters

- struct_ – Struct packing the values
- field_name_ – Field name in the struct
- type_ – Token type for JSON value corresponding to a primitive type. Must be one of: JSON_TOK_STRING for strings, JSON_TOK_NUMBER for numbers, JSON_TOK_TRUE (or JSON_TOK_FALSE) for booleans.

JSON_OBJ_DESCR_OBJECT(struct_, field_name_, sub_descr_)

Helper macro to declare a descriptor for an object value.

Here's an example of use:

```
struct nested {
    int foo;
    struct {
       int baz;
```

(continues on next page)

(continued from previous page)

```
    } bar;
};

struct json_obj_descr nested_bar[] = {
    { ... declare bar.baz descriptor ... },
};
struct json_obj_descr nested[] = {
    { ... declare foo descriptor ... },
    JSON_OBJ_DESCR_OBJECT(struct nested, bar, nested_bar),
};
```

Parameters

- `struct_` – Struct packing the values
- `field_name_` – Field name in the struct
- `sub_descr_` – Array of *json_obj_descr* describing the subobject

`JSON_OBJ_DESCR_ARRAY(struct_, field_name_, max_len_, len_field_, elem_type_)`

Helper macro to declare a descriptor for an array of primitives.

Here's an example of use:

```
struct example {
    int foo[10];
    size_t foo_len;
};

struct json_obj_descr array[] = {
     JSON_OBJ_DESCR_ARRAY(struct example, foo, 10, foo_len,
                          JSON_TOK_NUMBER)
};
```

Parameters

- `struct_` – Struct packing the values
- `field_name_` – Field name in the struct
- `max_len_` – Maximum number of elements in array
- `len_field_` – Field name in the struct for the number of elements in the array
- `elem_type_` – Element type, must be a primitive type

`JSON_OBJ_DESCR_OBJ_ARRAY(struct_, field_name_, max_len_, len_field_, elem_descr_, elem_descr_len_)`

Helper macro to declare a descriptor for an array of objects.

Here's an example of use:

```
struct person_height {
    const char *name;
    int height;
};
```

(continues on next page)

(continued from previous page)

```
struct people_heights {
    struct person_height heights[10];
    size_t heights_len;
};

struct json_obj_descr person_height_descr[] = {
     JSON_OBJ_DESCR_PRIM(struct person_height, name, JSON_TOK_STRING),
     JSON_OBJ_DESCR_PRIM(struct person_height, height, JSON_TOK_NUMBER),
};

struct json_obj_descr array[] = {
     JSON_OBJ_DESCR_OBJ_ARRAY(struct people_heights, heights, 10,
                              heights_len, person_height_descr,
                              ARRAY_SIZE(person_height_descr)),
};
```

Parameters

- `struct_` – Struct packing the values
- `field_name_` – Field name in the struct containing the array
- `max_len_` – Maximum number of elements in the array
- `len_field_` – Field name in the struct for the number of elements in the array
- `elem_descr_` – Element descriptor, pointer to a descriptor array
- `elem_descr_len_` – Number of elements in elem_descr_

`JSON_OBJ_DESCR_ARRAY_ARRAY(struct_, field_name_, max_len_, len_field_, elem_descr_, elem_descr_len_)`

Helper macro to declare a descriptor for an array of array.

Here's an example of use:

```
struct person_height {
    const char *name;
    int height;
};

struct person_heights_array {
    struct person_height heights;
}

struct people_heights {
    struct person_height_array heights[10];
    size_t heights_len;
};

struct json_obj_descr person_height_descr[] = {
    JSON_OBJ_DESCR_PRIM(struct person_height, name, JSON_TOK_STRING),
    JSON_OBJ_DESCR_PRIM(struct person_height, height, JSON_TOK_NUMBER),
};

struct json_obj_descr person_height_array_descr[] = {
    JSON_OBJ_DESCR_OBJECT(struct person_heights_array,
```

(continues on next page)

(continued from previous page)

```
                            heights, person_heigth_descr),
};

struct json_obj_descr array_array[] = {
     JSON_OBJ_DESCR_ARRAY_ARRAY(struct people_heights, heights, 10,
                                heights_len, person_height_array_descr,
                                ARRAY_SIZE(person_height_array_descr)),
};
```

Parameters

- `struct_` – Struct packing the values
- `field_name_` – Field name in the struct containing the array
- `max_len_` – Maximum number of elements in the array
- `len_field_` – Field name in the struct for the number of elements in the array
- `elem_descr_` – Element descriptor, pointer to a descriptor array
- `elem_descr_len_` – Number of elements in elem_descr_

`JSON_OBJ_DESCR_PRIM_NAMED(struct_, json_field_name_, struct_field_name_, type_)`

Variant of JSON_OBJ_DESCR_PRIM that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

See also:

JSON_OBJ_DESCR_PRIM

Parameters

- `struct_` – Struct packing the values.
- `json_field_name_` – String, field name in JSON strings
- `struct_field_name_` – Field name in the struct
- `type_` – Token type for JSON value corresponding to a primitive type.

`JSON_OBJ_DESCR_OBJECT_NAMED(struct_, json_field_name_, struct_field_name_, sub_descr_)`

Variant of JSON_OBJ_DESCR_OBJECT that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

See also:

JSON_OBJ_DESCR_OBJECT

Parameters

- `struct_` – Struct packing the values
- `json_field_name_` – String, field name in JSON strings
- `struct_field_name_` – Field name in the struct
- `sub_descr_` – Array of *json_obj_descr* describing the subobject

JSON_OBJ_DESCR_ARRAY_NAMED(struct_, json_field_name_, struct_field_name_, max_len_, len_field_, elem_type_)

Variant of JSON_OBJ_DESCR_ARRAY that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

See also:

JSON_OBJ_DESCR_ARRAY

Parameters

- struct_ – Struct packing the values
- json_field_name_ – String, field name in JSON strings
- struct_field_name_ – Field name in the struct
- max_len_ – Maximum number of elements in array
- len_field_ – Field name in the struct for the number of elements in the array
- elem_type_ – Element type, must be a primitive type

JSON_OBJ_DESCR_OBJ_ARRAY_NAMED(struct_, json_field_name_, struct_field_name_, max_len_, len_field_, elem_descr_, elem_descr_len_)

Variant of JSON_OBJ_DESCR_OBJ_ARRAY that can be used when the structure and JSON field names differ.

This is useful when the JSON field is not a valid C identifier.

Here's an example of use:

```
struct person_height {
    const char *name;
    int height;
};

struct people_heights {
    struct person_height heights[10];
    size_t heights_len;
};

struct json_obj_descr person_height_descr[] = {
     JSON_OBJ_DESCR_PRIM(struct person_height, name, JSON_TOK_STRING),
     JSON_OBJ_DESCR_PRIM(struct person_height, height, JSON_TOK_NUMBER),
};

struct json_obj_descr array[] = {
     JSON_OBJ_DESCR_OBJ_ARRAY_NAMED(struct people_heights,
                                    "people-heights", heights,
                                    10, heights_len,
                                    person_height_descr,
                                    ARRAY_SIZE(person_height_descr)),
};
```

Parameters

- struct_ – Struct packing the values

- `json_field_name_` – String, field name of the array in JSON strings
- `struct_field_name_` – Field name in the struct containing the array
- `max_len_` – Maximum number of elements in the array
- `len_field_` – Field name in the struct for the number of elements in the array
- `elem_descr_` – Element descriptor, pointer to a descriptor array
- `elem_descr_len_` – Number of elements in elem_descr_

Typedefs

typedef int (*`json_append_bytes_t`)(const char *bytes, size_t len, void *data)

Function pointer type to append bytes to a buffer while encoding JSON data.

Param bytes Contents to write to the output

Param len Number of bytes to append to output

Param data User-provided pointer

Return This callback function should return a negative number on error (which will be propagated to the return value of *json_obj_encode()*), or 0 on success.

Enums

enum `json_tokens`

Values:

enumerator `JSON_TOK_NONE` = '_'

enumerator `JSON_TOK_OBJECT_START` = '{'

enumerator `JSON_TOK_OBJECT_END` = '}'

enumerator `JSON_TOK_LIST_START` = '['

enumerator `JSON_TOK_LIST_END` = ']'

enumerator `JSON_TOK_STRING` = '"'

enumerator `JSON_TOK_COLON` = ':'

enumerator `JSON_TOK_COMMA` = ','

enumerator `JSON_TOK_NUMBER` = '0'

enumerator `JSON_TOK_TRUE` = 't'

enumerator `JSON_TOK_FALSE` = 'f'

enumerator JSON_TOK_NULL = 'n'

enumerator JSON_TOK_ERROR = '!'

enumerator JSON_TOK_EOF = '\0'

Functions

int json_obj_parse(char *json, size_t len, const struct *json_obj_descr* *descr, size_t descr_len, void *val)

Parses the JSON-encoded object pointer to by *json*, with size *len*, according to the descriptor pointed to by *descr*. Values are stored in a struct pointed to by *val*. Set up the descriptor like this:

struct s { int foo; char *bar; } struct *json_obj_descr* descr[] = { *JSON_OBJ_DESCR_PRIM(struct s, foo, JSON_TOK_NUMBER)*, *JSON_OBJ_DESCR_PRIM(struct s, bar, JSON_TOK_STRING)*, };

Since this parser is designed for machine-to-machine communications, some liberties were taken to simplify the design: (1) strings are not unescaped (but only valid escape sequences are accepted); (2) no UTF-8 validation is performed; and (3) only integer numbers are supported (no strtod() in the minimal libc).

Parameters

- json – Pointer to JSON-encoded value to be parsed
- len – Length of JSON-encoded value
- descr – Pointer to the descriptor array
- descr_len – Number of elements in the descriptor array. Must be less than 31 due to implementation detail reasons (if more fields are necessary, use two descriptors)
- val – Pointer to the struct to hold the decoded values

Returns < 0 if error, bitmap of decoded fields on success (bit 0 is set if first field in the descriptor has been properly decoded, etc).

ssize_t json_escape(char *str, size_t *len, size_t buf_size)

Escapes the string so it can be used to encode JSON objects.

Parameters

- str – The string to escape; the escape string is stored the buffer pointed to by this parameter
- len – Points to a size_t containing the size before and after the escaping process
- buf_size – The size of buffer str points to

Returns 0 if string has been escaped properly, or -ENOMEM if there was not enough space to escape the buffer

size_t json_calc_escaped_len(const char *str, size_t len)

Calculates the JSON-escaped string length.

Parameters

- str – The string to analyze
- len – String size

Returns The length str would have if it were escaped

ssize_t `json_calc_encoded_len`(const struct *json_obj_descr* *descr, size_t descr_len, const void *val)

Calculates the string length to fully encode an object.

Parameters

- `descr` – Pointer to the descriptor array
- `descr_len` – Number of elements in the descriptor array
- `val` – Struct holding the values

Returns Number of bytes necessary to encode the values if >0, an error code is returned.

int `json_obj_encode_buf`(const struct *json_obj_descr* *descr, size_t descr_len, const void *val, char *buffer, size_t buf_size)

Encodes an object in a contiguous memory location.

Parameters

- `descr` – Pointer to the descriptor array
- `descr_len` – Number of elements in the descriptor array
- `val` – Struct holding the values
- `buffer` – Buffer to store the JSON data
- `buf_size` – Size of buffer, in bytes, with space for the terminating NUL character

Returns 0 if object has been successfully encoded. A negative value indicates an error (as defined on errno.h).

int `json_arr_encode_buf`(const struct *json_obj_descr* *descr, const void *val, char *buffer, size_t buf_size)

Encodes an array in a contiguous memory location.

Parameters

- `descr` – Pointer to the descriptor array
- `val` – Struct holding the values
- `buffer` – Buffer to store the JSON data
- `buf_size` – Size of buffer, in bytes, with space for the terminating NUL character

Returns 0 if object has been successfully encoded. A negative value indicates an error (as defined on errno.h).

int `json_obj_encode`(const struct *json_obj_descr* *descr, size_t descr_len, const void *val, *json_append_bytes_t* append_bytes, void *data)

Encodes an object using an arbitrary writer function.

Parameters

- `descr` – Pointer to the descriptor array
- `descr_len` – Number of elements in the descriptor array
- `val` – Struct holding the values
- `append_bytes` – Function to append bytes to the output
- `data` – Data pointer to be passed to the append_bytes callback function.

Returns 0 if object has been successfully encoded. A negative value indicates an error.

int `json_arr_encode`(const struct *json_obj_descr* *descr, const void *val, *json_append_bytes_t* append_bytes, void *data)

Encodes an array using an arbitrary writer function.

Parameters

- `descr` – Pointer to the descriptor array
- `val` – Struct holding the values
- `append_bytes` – Function to append bytes to the output
- `data` – Data pointer to be passed to the append_bytes callback function.

Returns 0 if object has been successfully encoded. A negative value indicates an error.

struct `json_obj_descr`

#include <json.h>

JWT

JSON Web Tokens (JWT) are an open, industry standard [RFC 7519](https://tools.ietf.org/html/rfc7519) method for representing claims securely between two parties. Although JWT is fairly flexible, this API is limited to creating the simplistic tokens needed to authenticate with the Google Core IoT infrastructure.

group `jwt`

JSON Web Token (JWT)

Functions

int `jwt_init_builder`(struct *jwt_builder* *builder, char *buffer, size_t buffer_size)

Initialize the JWT builder.

Initialize the given JWT builder for the creation of a fresh token. The buffer size should at least be as long as JWT_BUILDER_MAX_SIZE returns.

Parameters

- `builder` – The builder to initialize.
- `buffer` – The buffer to write the token to.
- `buffer_size` – The size of this buffer. The token will be NULL terminated, which needs to be allowed for in this size.

Return values

- `0` – Success
- `-ENOSPC` – Buffer is insufficient to initialize

int `jwt_add_payload`(struct *jwt_builder* *builder, int32_t exp, int32_t iat, const char *aud)

add JWT primary payload.

int `jwt_sign`(struct *jwt_builder* *builder, const char *der_key, size_t der_key_len)

Sign the JWT token.

static inline size_t `jwt_payload_len`(struct *jwt_builder* *builder)

struct jwt_builder

#include <jwt.h> JWT data tracking.

JSON Web Tokens contain several sections, each encoded in base-64. This structure tracks the token as it is being built, including limits on the amount of available space. It should be initialized with jwt_init().

Public Members

char *base

The base of the buffer we are writing to.

char *buf

The place in this buffer where we are currently writing.

size_t len

The length remaining to write.

bool overflowed

Flag that is set if we try to write past the end of the buffer. If set, the token is not valid.

7.18 Data Structures

Zephyr provides a library of common general purpose data structures used within the kernel, but useful by application code in general. These include list and balanced tree structures for storing ordered data, and a ring buffer for managing "byte stream" data in a clean way.

Note that in general, the collections are implemented as "intrusive" data structures. The "node" data is the only struct used by the library code, and it does not store a pointer or other metadata to indicate what user data is "owned" by that node. Instead, the expectation is that the node will be itself embedded within a user-defined struct. Macros are provided to retrieve a user struct address from the embedded node pointer in a clean way. The purpose behind this design is to allow the collections to be used in contexts where dynamic allocation is disallowed (i.e. there is no need to allocate node objects because the memory is provided by the user).

Note also that these libraries are uniformly unsynchronized; access to them is not threadsafe by default. These are data structures, not synchronization primitives. The expectation is that any locking needed will be provided by the user.

7.18.1 Single-linked List

Zephyr provides a sys_slist_t type for storing simple singly-linked list data (i.e. data where each list element stores a pointer to the next element, but not the previous one). This supports constant-time access to the first (head) and last (tail) elements of the list, insertion before the head and after the tail of the list and constant time removal of the head. Removal of subsequent nodes requires access to the "previous" pointer and thus can only be performed in linear time by searching the list.

The sys_slist_t struct may be instantiated by the user in any accessible memory. It should be initialized with either *sys_slist_init()* or by static assignment from SYS_SLIST_STATIC_INIT before use. Its interior fields are opaque and should not be accessed by user code.

The end nodes of a list may be retrieved with *sys_slist_peek_head()* and *sys_slist_peek_tail()*, which will return NULL if the list is empty, otherwise a pointer to a sys_snode_t struct.

The `sys_snode_t` struct represents the data to be inserted. In general, it is expected to be allocated/controlled by the user, usually embedded within a struct which is to be added to the list. The container struct pointer may be retrieved from a list node using *SYS_SLIST_CONTAINER*, passing it the struct name of the containing struct and the field name of the node. Internally, the `sys_snode_t` struct contains only a next pointer, which may be accessed with *sys_slist_peek_next()*.

Lists may be modified by adding a single node at the head or tail with *sys_slist_prepend()* and *sys_slist_append()*. They may also have a node added to an interior point with *sys_slist_insert()*, which inserts a new node after an existing one. Similarly *sys_slist_remove()* will remove a node given a pointer to its predecessor. These operations are all constant time.

Convenience routines exist for more complicated modifications to a list. *sys_slist_merge_slist()* will append an entire list to an existing one. *sys_slist_append_list()* will append a bounded subset of an existing list in constant time. And *sys_slist_find_and_remove()* will search a list (in linear time) for a given node and remove it if present.

Finally the slist implementation provides a set of "for each" macros that allows for iterating over a list in a natural way without needing to manually traverse the next pointers. *SYS_SLIST_FOR_EACH_NODE* will enumerate every node in a list given a local variable to store the node pointer. *SYS_SLIST_FOR_EACH_NODE_SAFE* behaves similarly, but has a more complicated implementation that requires an extra scratch variable for storage and allows the user to delete the iterated node during the iteration. Each of those macros also exists in a "container" variant (*SYS_SLIST_FOR_EACH_CONTAINER* and *SYS_SLIST_FOR_EACH_CONTAINER_SAFE*) which assigns a local variable of a type that matches the user's container struct and not the node struct, performing the required offsets internally. And *SYS_SLIST_ITERATE_FROM_NODE* exists to allow for enumerating a node and all its successors only, without inspecting the earlier part of the list.

Single-linked List Internals

The slist code is designed to be minimal and conventional. Internally, a `sys_slist_t` struct is nothing more than a pair of "head" and "tail" pointer fields. And a `sys_snode_t` stores only a single "next" pointer.

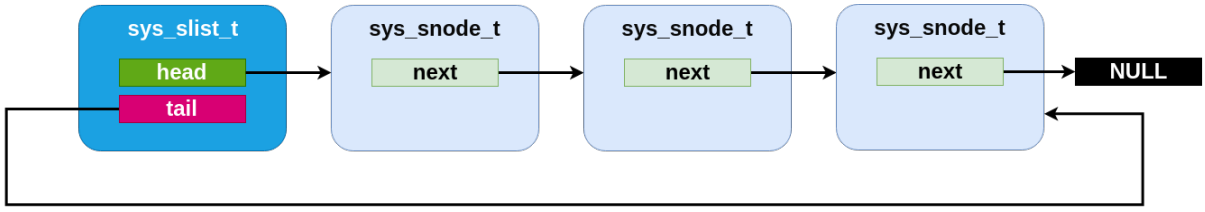


Fig. 3: An slist containing three elements.

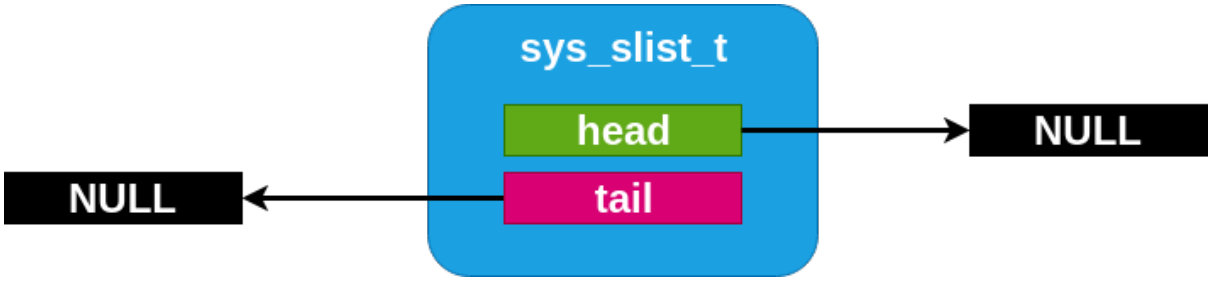


Fig. 4: An empty slist

The specific implementation of the list code, however, is done with an internal "Z_GENLIST" template API which allows for extracting those fields from arbitrary structures and emits an arbitrarily named set of functions. This allows for implementing more complicated single-linked list variants using the same basic primitives. The genlist implementor is responsible for a custom implementation of the primitive

operations only: an “init” step for each struct, and a “get” and “set” primitives for each of head, tail and next pointers on their relevant structs. These inline functions are passed as parameters to the genlist macro expansion.

Only one such variant, sflist, exists in Zephyr at the moment.

Flagged List

The `sys_sflist_t` is implemented using the described genlist template API. With the exception of symbol naming (“sflist” instead of “slist”) and the additional API described next, it operates in all ways identically to the slist API.

It adds the ability to associate exactly two bits of user defined “flags” with each list node. These can be accessed and modified with *sys_sfnode_flags_get()* and *sys_sfnode_flags_get()*. Internally, the flags are stored unioned with the bottom bits of the next pointer and incur no SRAM storage overhead when compared with the simpler slist code.

Single-linked List API Reference

group `single-linked-list_apis`

Defines

`SYS_SLIST_FOR_EACH_NODE(__sl, __sn)`

Provide the primitive to iterate on a list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_NODE(l, n) {
    <user code>
}
```

This and other SYS_SLIST_*() macros are not thread safe.

Parameters

- `__sl` – A pointer on a sys_slist_t to iterate on
- `__sn` – A sys_snode_t pointer to peek each node of the list

`SYS_SLIST_ITERATE_FROM_NODE(__sl, __sn)`

Provide the primitive to iterate on a list, from a node in the list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_ITERATE_FROM_NODE(l, n) {
    <user code>
}
```

Like *SYS_SLIST_FOR_EACH_NODE()*, but __dn already contains a node in the list where to start searching for the next entry from. If NULL, it starts from the head.

This and other SYS_SLIST_*() macros are not thread safe.

Parameters

- `__sl` – A pointer on a sys_slist_t to iterate on

- __sn – A sys_snode_t pointer to peek each node of the list it contains the starting node, or NULL to start from the head

SYS_SLIST_FOR_EACH_NODE_SAFE(__sl, __sn, __sns)

Provide the primitive to safely iterate on a list Note: __sn can be removed, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_NODE_SAFE(l, n, s) {
    <user code>
}
```

This and other SYS_SLIST_*() macros are not thread safe.

Parameters

- __sl – A pointer on a sys_slist_t to iterate on
- __sn – A sys_snode_t pointer to peek each node of the list
- __sns – A sys_snode_t pointer for the loop to run safely

SYS_SLIST_CONTAINER(__ln, __cn, __n)

SYS_SLIST_PEEK_HEAD_CONTAINER(__sl, __cn, __n)

SYS_SLIST_PEEK_TAIL_CONTAINER(__sl, __cn, __n)

SYS_SLIST_PEEK_NEXT_CONTAINER(__cn, __n)

SYS_SLIST_FOR_EACH_CONTAINER(__sl, __cn, __n)

Provide the primitive to iterate on a list under a container Note: the loop is unsafe and thus __cn should not be detached.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_CONTAINER(l, c, n) {
    <user code>
}
```

Parameters

- __sl – A pointer on a sys_slist_t to iterate on
- __cn – A pointer to peek each entry of the list
- __n – The field name of sys_node_t within the container struct

SYS_SLIST_FOR_EACH_CONTAINER_SAFE(__sl, __cn, __cns, __n)

Provide the primitive to safely iterate on a list under a container Note: __cn can be detached, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SLIST_FOR_EACH_NODE_SAFE(l, c, cn, n) {
    <user code>
}
```

Parameters

- __sl – A pointer on a sys_slist_t to iterate on
- __cn – A pointer to peek each entry of the list
- __cns – A pointer for the loop to run safely

- `__n` – The field name of sys_node_t within the container struct

`SYS_SLIST_STATIC_INIT(ptr_to_list)`

Functions

static inline void `sys_slist_init(sys_slist_t *list)`

Initialize a list.

Parameters

- `list` – A pointer on the list to initialize

static inline sys_snode_t *`sys_slist_peek_head(sys_slist_t *list)`

Peek the first node from the list.

Parameters

- `list` – A point on the list to peek the first node from

Returns A pointer on the first node of the list (or NULL if none)

static inline sys_snode_t *`sys_slist_peek_tail(sys_slist_t *list)`

Peek the last node from the list.

Parameters

- `list` – A point on the list to peek the last node from

Returns A pointer on the last node of the list (or NULL if none)

static inline bool `sys_slist_is_empty(sys_slist_t *list)`

Test if the given list is empty.

Parameters

- `list` – A pointer on the list to test

Returns a boolean, true if it's empty, false otherwise

static inline sys_snode_t *`sys_slist_peek_next_no_check(sys_snode_t *node)`

Peek the next node from current node, node is not NULL.

Faster then *sys_slist_peek_next()* if node is known not to be NULL.

Parameters

- `node` – A pointer on the node where to peek the next node

Returns a pointer on the next node (or NULL if none)

static inline sys_snode_t *`sys_slist_peek_next(sys_snode_t *node)`

Peek the next node from current node.

Parameters

- `node` – A pointer on the node where to peek the next node

Returns a pointer on the next node (or NULL if none)

static inline void `sys_slist_prepend(sys_slist_t *list, sys_snode_t *node)`

Prepend a node to the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect

- `node` – A pointer on the node to prepend

static inline void `sys_slist_append(sys_slist_t *list, sys_snode_t *node)`

Append a node to the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `node` – A pointer on the node to append

static inline void `sys_slist_append_list(sys_slist_t *list, void *head, void *tail)`

Append a list to the given list.

Append a singly-linked, NULL-terminated list consisting of nodes containing the pointer to the next node as the first element of a node, to *list*. This and other sys_slist_*() functions are not thread safe.

FIXME: Why are the element parameters void *?

Parameters

- `list` – A pointer on the list to affect
- `head` – A pointer to the first element of the list to append
- `tail` – A pointer to the last element of the list to append

static inline void `sys_slist_merge_slist(sys_slist_t *list, sys_slist_t *list_to_append)`

merge two slists, appending the second one to the first

When the operation is completed, the appending list is empty. This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `list_to_append` – A pointer to the list to append.

static inline void `sys_slist_insert(sys_slist_t *list, sys_snode_t *prev, sys_snode_t *node)`

Insert a node to the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `prev` – A pointer on the previous node
- `node` – A pointer on the node to insert

static inline sys_snode_t *`sys_slist_get_not_empty(sys_slist_t *list)`

Fetch and remove the first node of the given list.

List must be known to be non-empty. This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect

Returns A pointer to the first node of the list

static inline sys_snode_t *`sys_slist_get(sys_slist_t *list)`

Fetch and remove the first node of the given list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect

Returns A pointer to the first node of the list (or NULL if empty)

static inline void `sys_slist_remove`(sys_slist_t *list, sys_snode_t *prev_node, sys_snode_t *node)

Remove a node.

This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `prev_node` – A pointer on the previous node (can be NULL, which means the node is the list's head)
- `node` – A pointer on the node to remove

static inline bool `sys_slist_find_and_remove`(sys_slist_t *list, sys_snode_t *node)

Find and remove a node from a list.

This and other sys_slist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `node` – A pointer on the node to remove from the list

Returns true if node was removed

Flagged List API Reference

group `flagged-single-linked-list_apis`

Defines

`SYS_SFLIST_FOR_EACH_NODE(__sl, __sn)`

Provide the primitive to iterate on a list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_NODE(l, n) {
    <user code>
}
```

This and other SYS_SFLIST_*() macros are not thread safe.

Parameters

- `__sl` – A pointer on a sys_sflist_t to iterate on
- `__sn` – A sys_sfnode_t pointer to peek each node of the list

`SYS_SFLIST_ITERATE_FROM_NODE(__sl, __sn)`

Provide the primitive to iterate on a list, from a node in the list Note: the loop is unsafe and thus __sn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_ITERATE_FROM_NODE(l, n) {
    <user code>
}
```

Like *SYS_SFLIST_FOR_EACH_NODE()*, but __dn already contains a node in the list where to start searching for the next entry from. If NULL, it starts from the head.

This and other SYS_SFLIST_*() macros are not thread safe.

Parameters

- `__sl` – A pointer on a sys_sflist_t to iterate on
- `__sn` – A sys_sfnode_t pointer to peek each node of the list it contains the starting node, or NULL to start from the head

`SYS_SFLIST_FOR_EACH_NODE_SAFE(__sl, __sn, __sns)`

Provide the primitive to safely iterate on a list Note: __sn can be removed, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_NODE_SAFE(l, n, s) {
    <user code>
}
```

This and other SYS_SFLIST_*() macros are not thread safe.

Parameters

- `__sl` – A pointer on a sys_sflist_t to iterate on
- `__sn` – A sys_sfnode_t pointer to peek each node of the list
- `__sns` – A sys_sfnode_t pointer for the loop to run safely

`SYS_SFLIST_CONTAINER(__ln, __cn, __n)`

`SYS_SFLIST_PEEK_HEAD_CONTAINER(__sl, __cn, __n)`

`SYS_SFLIST_PEEK_TAIL_CONTAINER(__sl, __cn, __n)`

`SYS_SFLIST_PEEK_NEXT_CONTAINER(__cn, __n)`

`SYS_SFLIST_FOR_EACH_CONTAINER(__sl, __cn, __n)`

Provide the primitive to iterate on a list under a container Note: the loop is unsafe and thus __cn should not be detached.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_CONTAINER(l, c, n) {
    <user code>
}
```

Parameters

- `__sl` – A pointer on a sys_sflist_t to iterate on
- `__cn` – A pointer to peek each entry of the list
- `__n` – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_FOR_EACH_CONTAINER_SAFE(__sl, __cn, __cns, __n)

Provide the primitive to safely iterate on a list under a container Note: __cn can be detached, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_SFLIST_FOR_EACH_NODE_SAFE(l, c, cn, n) {
    <user code>
}
```

Parameters

- __sl – A pointer on a sys_sflist_t to iterate on
- __cn – A pointer to peek each entry of the list
- __cns – A pointer for the loop to run safely
- __n – The field name of sys_sfnode_t within the container struct

SYS_SFLIST_STATIC_INIT(ptr_to_list)

SYS_SFLIST_FLAGS_MASK

Functions

static inline void sys_sflist_init(sys_sflist_t *list)

Initialize a list.

Parameters

- list – A pointer on the list to initialize

static inline uint8_t sys_sfnode_flags_get(sys_sfnode_t *node)

Fetch flags value for a particular sfnode.

Parameters

- node – A pointer to the node to fetch flags from

Returns The value of flags, which will be between 0 and 3

static inline sys_sfnode_t *sys_sflist_peek_head(sys_sflist_t *list)

Peek the first node from the list.

Parameters

- list – A point on the list to peek the first node from

Returns A pointer on the first node of the list (or NULL if none)

static inline sys_sfnode_t *sys_sflist_peek_tail(sys_sflist_t *list)

Peek the last node from the list.

Parameters

- list – A point on the list to peek the last node from

Returns A pointer on the last node of the list (or NULL if none)

static inline void `sys_sfnode_init`(sys_sfnode_t *node, uint8_t flags)

Initialize an sflist node.

Set an initial flags value for this slist node, which can be a value between 0 and 3. These flags will persist even if the node is moved around within a list, removed, or transplanted to a different slist.

This is ever so slightly faster than *sys_sfnode_flags_set()* and should only be used on a node that hasn't been added to any list.

Parameters

- `node` – A pointer to the node to set the flags on
- `flags` – A value between 0 and 3 to set the flags value

static inline void `sys_sfnode_flags_set`(sys_sfnode_t *node, uint8_t flags)

Set flags value for an sflist node.

Set a flags value for this slist node, which can be a value between 0 and 3. These flags will persist even if the node is moved around within a list, removed, or transplanted to a different slist.

Parameters

- `node` – A pointer to the node to set the flags on
- `flags` – A value between 0 and 3 to set the flags value

static inline bool `sys_sflist_is_empty`(sys_sflist_t *list)

Test if the given list is empty.

Parameters

- `list` – A pointer on the list to test

Returns a boolean, true if it's empty, false otherwise

static inline sys_sfnode_t *`sys_sflist_peek_next_no_check`(sys_sfnode_t *node)

Peek the next node from current node, node is not NULL.

Faster then *sys_sflist_peek_next()* if node is known not to be NULL.

Parameters

- `node` – A pointer on the node where to peek the next node

Returns a pointer on the next node (or NULL if none)

static inline sys_sfnode_t *`sys_sflist_peek_next`(sys_sfnode_t *node)

Peek the next node from current node.

Parameters

- `node` – A pointer on the node where to peek the next node

Returns a pointer on the next node (or NULL if none)

static inline void `sys_sflist_prepend`(sys_sflist_t *list, sys_sfnode_t *node)

Prepend a node to the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `node` – A pointer on the node to prepend

static inline void sys_sflist_append(sys_sflist_t *list, sys_sfnode_t *node)

Append a node to the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- list – A pointer on the list to affect
- node – A pointer on the node to append

static inline void sys_sflist_append_list(sys_sflist_t *list, void *head, void *tail)

Append a list to the given list.

Append a singly-linked, NULL-terminated list consisting of nodes containing the pointer to the next node as the first element of a node, to *list*. This and other sys_sflist_*() functions are not thread safe.

FIXME: Why are the element parameters void *?

Parameters

- list – A pointer on the list to affect
- head – A pointer to the first element of the list to append
- tail – A pointer to the last element of the list to append

static inline void sys_sflist_merge_sflist(sys_sflist_t *list, sys_sflist_t *list_to_append)

merge two sflists, appending the second one to the first

When the operation is completed, the appending list is empty. This and other sys_sflist_*() functions are not thread safe.

Parameters

- list – A pointer on the list to affect
- list_to_append – A pointer to the list to append.

static inline void sys_sflist_insert(sys_sflist_t *list, sys_sfnode_t *prev, sys_sfnode_t *node)

Insert a node to the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- list – A pointer on the list to affect
- prev – A pointer on the previous node
- node – A pointer on the node to insert

static inline sys_sfnode_t *sys_sflist_get_not_empty(sys_sflist_t *list)

Fetch and remove the first node of the given list.

List must be known to be non-empty. This and other sys_sflist_*() functions are not thread safe.

Parameters

- list – A pointer on the list to affect

Returns A pointer to the first node of the list

static inline sys_sfnode_t *sys_sflist_get(sys_sflist_t *list)

Fetch and remove the first node of the given list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect

Returns A pointer to the first node of the list (or NULL if empty)

static inline void `sys_sflist_remove`(sys_sflist_t *list, sys_sfnode_t *prev_node, sys_sfnode_t *node)

Remove a node.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `prev_node` – A pointer on the previous node (can be NULL, which means the node is the list's head)
- `node` – A pointer on the node to remove

static inline bool `sys_sflist_find_and_remove`(sys_sflist_t *list, sys_sfnode_t *node)

Find and remove a node from a list.

This and other sys_sflist_*() functions are not thread safe.

Parameters

- `list` – A pointer on the list to affect
- `node` – A pointer on the node to remove from the list

Returns true if node was removed

7.18.2 Double-linked List

Similar to the single-linked list in many respects, Zephyr includes a double-linked implementation. This provides the same algorithmic behavior for all the existing slist operations, but also allows for constant-time removal and insertion (at all points: before or after the head, tail or any internal node). To do this, the list stores two pointers per node, and thus has somewhat higher runtime code and memory space needs.

A *sys_dlist_t* struct may be instantiated by the user in any accessible memory. It must be initialized with *sys_dlist_init()* or *SYS_DLIST_STATIC_INIT* before use. The *sys_dnode_t* struct is expected to be provided by the user for any nodes addded to the list (typically embedded within the struct to be tracked, as described above). It must be initialized in zeroed/bss memory or with *sys_dnode_init()* before use.

Primitive operations may retrieve the head/tail of a list and the next/prev pointers of a node with *sys_dlist_peek_head()*, *sys_dlist_peek_tail()*, *sys_dlist_peek_next()* and *sys_dlist_peek_prev()*. These can all return NULL where appropriate (i.e. for empty lists, or nodes at the endpoints of the list).

A dlist can be modified in constant time by removing a node with *sys_dlist_remove()*, by adding a node to the head or tail of a list with *sys_dlist_prepend()* and *sys_dlist_append()*, or by inserting a node before an existing node with *sys_dlist_insert()*.

As for slist, each node in a dlist can be processed in a natural code block style using *SYS_DLIST_FOR_EACH_NODE*. This macro also exists in a "FROM_NODE" form which allows for iterating from a known starting point, a "SAFE" variant that allows for removing the node being inspected within the code block, a "CONTAINER" style that provides the pointer to a containing struct instead of the raw node, and a "CONTAINER_SAFE" variant that provides both properties.

Convenience utilities provided by dlist include *sys_dlist_insert_at()*, which inserts a node that linearly searches through a list to find the right insertion point, which is provided by the user as a C callback function pointer, and *sys_dnode_is_linked()*, which will affirmatively return whether or not a node is currently linked into a dlist or not (via an implementation that has zero overhead vs. the normal list processing).

Double-linked List Internals

Internally, the dlist implementation is minimal: the *sys_dlist_t* struct contains "head" and "tail" pointer fields, the *sys_dnode_t* contains "prev" and "next" pointers, and no other data is stored. But in practice the two structs are internally identical, and the list struct is inserted as a node into the list itself. This allows for a very clean symmetry of operations:

- An empty list has backpointers to itself in the list struct, which can be trivially detected.
- The head and tail of the list can be detected by comparing the prev/next pointers of a node vs. the list struct address.
- An insertion or deletion never needs to check for the special case of inserting at the head or tail. There are never any NULL pointers within the list to be avoided. Exactly the same operations are run, without tests or branches, for all list modification primitives.

Effectively, a dlist of N nodes can be thought of as a "ring" of "N+1" nodes, where one node represents the list tracking struct.

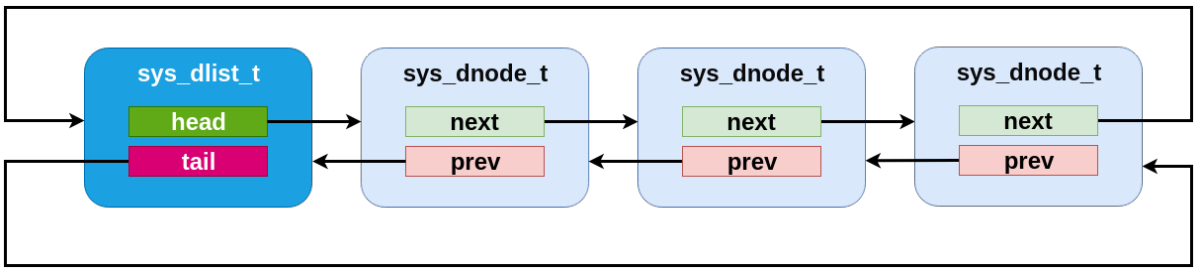


Fig. 5: A dlist containing three elements. Note that the list struct appears as a fourth "element" in the list.

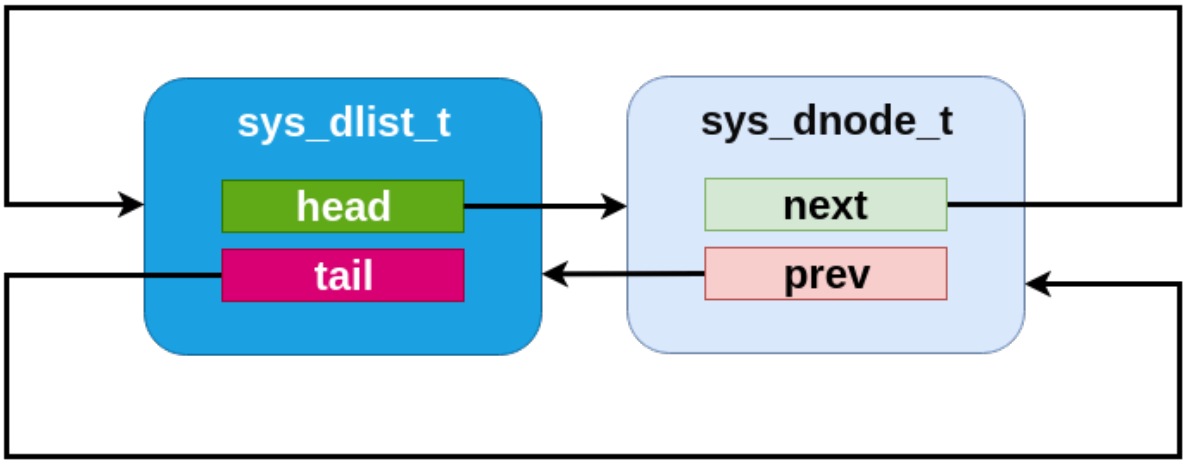


Fig. 6: An dlist containing just one element.

Doubly-linked List API Reference

group `doubly-linked-list_apis`

Defines

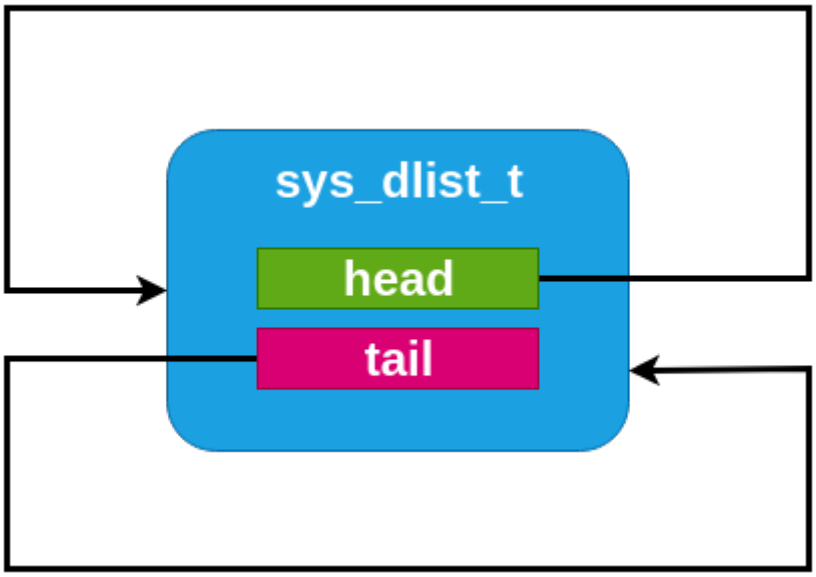


Fig. 7: An empty dlist.

SYS_DLIST_FOR_EACH_NODE(__dl, __dn)

Provide the primitive to iterate on a list Note: the loop is unsafe and thus __dn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_NODE(l, n) {
    <user code>
}
```

This and other SYS_DLIST_*() macros are not thread safe.

Parameters

- __dl – A pointer on a sys_dlist_t to iterate on
- __dn – A sys_dnode_t pointer to peek each node of the list

SYS_DLIST_ITERATE_FROM_NODE(__dl, __dn)

Provide the primitive to iterate on a list, from a node in the list Note: the loop is unsafe and thus __dn should not be removed.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_ITERATE_FROM_NODE(l, n) {
    <user code>
}
```

Like *SYS_DLIST_FOR_EACH_NODE()*, but __dn already contains a node in the list where to start searching for the next entry from. If NULL, it starts from the head.

This and other SYS_DLIST_*() macros are not thread safe.

Parameters

- __dl – A pointer on a sys_dlist_t to iterate on
- __dn – A sys_dnode_t pointer to peek each node of the list; it contains the starting node, or NULL to start from the head

SYS_DLIST_FOR_EACH_NODE_SAFE(__dl, __dn, __dns)

Provide the primitive to safely iterate on a list Note: __dn can be removed, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_NODE_SAFE(l, n, s) {
    <user code>
}
```

This and other SYS_DLIST_*() macros are not thread safe.

Parameters

- __dl – A pointer on a sys_dlist_t to iterate on
- __dn – A sys_dnode_t pointer to peek each node of the list
- __dns – A sys_dnode_t pointer for the loop to run safely

SYS_DLIST_CONTAINER(__dn, __cn, __n)

SYS_DLIST_PEEK_HEAD_CONTAINER(__dl, __cn, __n)

SYS_DLIST_PEEK_NEXT_CONTAINER(__dl, __cn, __n)

SYS_DLIST_FOR_EACH_CONTAINER(__dl, __cn, __n)

Provide the primitive to iterate on a list under a container Note: the loop is unsafe and thus __cn should not be detached.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_CONTAINER(l, c, n) {
    <user code>
}
```

Parameters

- __dl – A pointer on a sys_dlist_t to iterate on
- __cn – A pointer to peek each entry of the list
- __n – The field name of sys_dnode_t within the container struct

SYS_DLIST_FOR_EACH_CONTAINER_SAFE(__dl, __cn, __cns, __n)

Provide the primitive to safely iterate on a list under a container Note: __cn can be detached, it will not break the loop.

User *MUST* add the loop statement curly braces enclosing its own code:

```
SYS_DLIST_FOR_EACH_CONTAINER_SAFE(l, c, cn, n) {
    <user code>
}
```

Parameters

- __dl – A pointer on a sys_dlist_t to iterate on
- __cn – A pointer to peek each entry of the list
- __cns – A pointer for the loop to run safely
- __n – The field name of sys_dnode_t within the container struct

SYS_DLIST_STATIC_INIT(ptr_to_list)

Typedefs

typedef struct _dnode `sys_dlist_t`

typedef struct _dnode `sys_dnode_t`

Functions

static inline void `sys_dlist_init`(*sys_dlist_t* *list)

initialize list to its empty state

Parameters

- `list` – the doubly-linked list

Returns N/A

static inline void `sys_dnode_init`(*sys_dnode_t* *node)

initialize node to its state when not in a list

Parameters

- `node` – the node

Returns N/A

static inline bool `sys_dnode_is_linked`(const *sys_dnode_t* *node)

check if a node is a member of any list

Parameters

- `node` – the node

Returns true if node is linked into a list, false if it is not

static inline bool `sys_dlist_is_head`(*sys_dlist_t* *list, *sys_dnode_t* *node)

check if a node is the list's head

Parameters

- `list` – the doubly-linked list to operate on
- `node` – the node to check

Returns true if node is the head, false otherwise

static inline bool `sys_dlist_is_tail`(*sys_dlist_t* *list, *sys_dnode_t* *node)

check if a node is the list's tail

Parameters

- `list` – the doubly-linked list to operate on
- `node` – the node to check

Returns true if node is the tail, false otherwise

static inline bool `sys_dlist_is_empty`(*sys_dlist_t* *list)

check if the list is empty

Parameters

- `list` – the doubly-linked list to operate on

Returns true if empty, false otherwise

static inline bool sys_dlist_has_multiple_nodes(*sys_dlist_t* *list)

check if more than one node present

This and other sys_dlist_*() functions are not thread safe.

Parameters

- list – the doubly-linked list to operate on

Returns true if multiple nodes, false otherwise

static inline *sys_dnode_t* *sys_dlist_peek_head(*sys_dlist_t* *list)

get a reference to the head item in the list

Parameters

- list – the doubly-linked list to operate on

Returns a pointer to the head element, NULL if list is empty

static inline *sys_dnode_t* *sys_dlist_peek_head_not_empty(*sys_dlist_t* *list)

get a reference to the head item in the list

The list must be known to be non-empty.

Parameters

- list – the doubly-linked list to operate on

Returns a pointer to the head element

static inline *sys_dnode_t* *sys_dlist_peek_next_no_check(*sys_dlist_t* *list, *sys_dnode_t* *node)

get a reference to the next item in the list, node is not NULL

Faster than *sys_dlist_peek_next()* if node is known not to be NULL.

Parameters

- list – the doubly-linked list to operate on
- node – the node from which to get the next element in the list

Returns a pointer to the next element from a node, NULL if node is the tail

static inline *sys_dnode_t* *sys_dlist_peek_next(*sys_dlist_t* *list, *sys_dnode_t* *node)

get a reference to the next item in the list

Parameters

- list – the doubly-linked list to operate on
- node – the node from which to get the next element in the list

Returns a pointer to the next element from a node, NULL if node is the tail or NULL (when node comes from reading the head of an empty list).

static inline *sys_dnode_t* *sys_dlist_peek_prev_no_check(*sys_dlist_t* *list, *sys_dnode_t* *node)

get a reference to the previous item in the list, node is not NULL

Faster than *sys_dlist_peek_prev()* if node is known not to be NULL.

Parameters

- list – the doubly-linked list to operate on
- node – the node from which to get the previous element in the list

Returns a pointer to the previous element from a node, NULL if node is the tail

static inline *sys_dnode_t* *sys_dlist_peek_prev(*sys_dlist_t* *list, *sys_dnode_t* *node)

get a reference to the previous item in the list

Parameters

- `list` – the doubly-linked list to operate on
- `node` – the node from which to get the previous element in the list

Returns a pointer to the previous element from a node, NULL if node is the tail or NULL (when node comes from reading the head of an empty list).

static inline *sys_dnode_t* *sys_dlist_peek_tail(*sys_dlist_t* *list)

get a reference to the tail item in the list

Parameters

- `list` – the doubly-linked list to operate on

Returns a pointer to the tail element, NULL if list is empty

static inline void sys_dlist_append(*sys_dlist_t* *list, *sys_dnode_t* *node)

add node to tail of list

This and other sys_dlist_*() functions are not thread safe.

Parameters

- `list` – the doubly-linked list to operate on
- `node` – the element to append

Returns N/A

static inline void sys_dlist_prepend(*sys_dlist_t* *list, *sys_dnode_t* *node)

add node to head of list

This and other sys_dlist_*() functions are not thread safe.

Parameters

- `list` – the doubly-linked list to operate on
- `node` – the element to append

Returns N/A

static inline void sys_dlist_insert(*sys_dnode_t* *successor, *sys_dnode_t* *node)

Insert a node into a list.

Insert a node before a specified node in a dlist.

Parameters

- `successor` – the position before which "node" will be inserted
- `node` – the element to insert

static inline void sys_dlist_insert_at(*sys_dlist_t* *list, *sys_dnode_t* *node, int (*cond)(*sys_dnode_t* *node, void *data), void *data)

insert node at position

Insert a node in a location depending on a external condition. The cond() function checks if the node is to be inserted *before* the current node against which it is checked. This and other sys_dlist_*() functions are not thread safe.

Parameters

- `list` – the doubly-linked list to operate on
- `node` – the element to insert

- `cond` – a function that determines if the current node is the correct insert point
- `data` – parameter to cond()

Returns N/A

static inline void `sys_dlist_remove`(*sys_dnode_t* *node)

remove a specific node from a list

The list is implicit from the node. The node must be part of a list. This and other sys_dlist_*() functions are not thread safe.

Parameters

- `node` – the node to remove

Returns N/A

static inline *sys_dnode_t* *`sys_dlist_get`(*sys_dlist_t* *list)

get the first node in a list

This and other sys_dlist_*() functions are not thread safe.

Parameters

- `list` – the doubly-linked list to operate on

Returns the first node in the list, NULL if list is empty

7.18.3 Multi Producer Single Consumer Packet Buffer

A *Multi Producer Single Consumer Packet Buffer (MPSC_PBUF)* is a circular buffer, whose contents are stored in first-in-first-out order. Variable size packets are stored in the buffer. Packet buffer works under assumption that there is a single context that consumes the data. However, it is possible that another context may interfere to flush the data and never come back (panic case). Packet is produced in two steps: first requested amount of data is allocated, producer fills the data and commits it. Consuming a packet is also performed in two steps: consumer claims the packet, gets pointer to it and length and later on packet is freed. This approach reduces memory copying.

A *MPSC Packet Buffer* has the following key properties:

- Allocate, commit scheme used for packet producing.
- Claim, free scheme used for packet consuming.
- Allocator ensures that continue memory of requested length is allocated.
- Following policies can be applied when requested space cannot be allocated:
  - **Overwrite** - oldest entries are dropped until requested amount of memory can be allocated. For each dropped packet user callback is called.
  - **No overwrite** - When requested amount of space cannot be allocated, allocation fails.
- Dedicated, optimized API for storing short packets.
- Allocation with timeout.

Internals

Each packet in the buffer contains `MPSC_PBUF` specific header which is used for internal management. Header consists of 2 bit flags. In order to optimize memory usage, header can be added on top of the user header using `MPSC_PBUF_HDR` and remaining bits in the first word can be application specific. Header consists of following flags:

- valid - bit set to one when packet contains valid user packet

- busy - bit set when packet is being consumed (claimed but not free)

Header state:

valid	busy	description
0	0	space is free
1	0	valid packet
1	1	claimed valid packet
0	1	internal skip packet

Packet buffer space contains free space, valid user packets and internal skip packets. Internal skip packets indicates padding, e.g. at the of the buffer.

Allocation Using pairs for read and write indexes, available space is determined. If space can be allocated, temporary write index is moved and pointer to a space witing buffer is returned. Packet header is reset. If allocation required wrapping of the write index, a skip packet is added to the end of buffer. If space cannot be allocated and overwrite is disabled then `NULL` pointer is returned or context blocks if allocation was with timeout.

Allocation with overwrite If overwrite is enabled, oldest packets are dropped until requested amount of space can be allocated. When packets are dropped `busy` flag is checked in the header to ensure that currently consumed packet is not overwritten. In that case, skip packet is added before busy packet and packets following the busy packet are dropped. When busy packet is being freed, such situation is detected and packet is converted to skip packet to avoid double processing.

Usage

Packet header definition Packet header details can be found in include/sys/mpsc_packet.h. API functions can be found in include/sys/mpsc_pbuf.h. Headers are split to avoid include spam when declaring the packet.

User header structure must start with internal header:

```
#include <sys/mpsc_packet.h>

struct foo_header {
        MPSC_PBUF_HDR;
        uint32_t length: 32 - MPSC_PBUF_HDR_BITS;
};
```

Packet buffer configuration Configuration structure contains buffer details, configuration flags and callbacks. Following callbacks are used by the packet buffer:

- Drop notification - callback called whenever a packet is dropped due to overwrite.
- Get packet length - callback to determine packet length

Packet producing Standard, two step method:

```
foo_packet *packet = mpsc_pbuf_alloc(buffer, len, K_NO_WAIT);

fill_data(packet);

mpsc_pbuf_commit(buffer, packet);
```

Performance optimized storing of small packets:

- 32 bit word packet
- 32 bit word with pointer packet

Note that since packets are written by value, they should already contain `valid` bit set in the header.

```
mpsc_pbuf_put_word(buffer, data);
mpsc_pbuf_put_word_ext(buffer, data, ptr);
```

Packet consuming Two step method:

```
foo_packet *packet = mpsc_pbuf_claim(buffer);

process(packet);

mpsc_pbuf_free(buffer, packet);
```

7.18.4 Balanced Red/Black Tree

For circumstances where sorted containers may become large at runtime, a list becomes problematic due to algorithmic costs of searching it. For these situations, Zephyr provides a balanced tree implementation which has runtimes on search and removal operations bounded at O(log2(N)) for a tree of size N. This is implemented using a conventional red/black tree as described by multiple academic sources.

The *rbtree* tracking struct for a rbtree may be initialized anywhere in user accessible memory. It should contain only zero bits before first use. No specific initialization API is needed or required.

Unlike a list, where position is explicit, the ordering of nodes within an rbtree must be provided as a predicate function by the user. A function of type *rb_lessthan_t()* should be assigned to the `lessthan_fn` field of the :c:struct`rbtree` struct before any tree operations are attempted. This function should, as its name suggests, return a boolean True value if the first node argument is "less than" the second in the ordering desired by the tree. Note that "equal" is not allowed, nodes within a tree must have a single fixed order for the algorithm to work correctly.

As with the slist and dlist containers, nodes within an rbtree are represented as a `rbnode` structure which exists in user-managed memory, typically embedded within the the data structure being tracked in the tree. Unlike the list code, the data within an rbnode is entirely opaque. It is not possible for the user to extract the binary tree topology and "manually" traverse the tree as it is for a list.

Nodes can be inserted into a tree with *rb_insert()* and removed with *rb_remove()*. Access to the "first" and "last" nodes within a tree (in the sense of the order defined by the comparison function) is provided by *rb_get_min()* and *rb_get_max()*. There is also a predicate, *rb_contains()*, which returns a boolean True if the provided node pointer exists as an element within the tree. As described above, all of these routines are guaranteed to have at most log time complexity in the size of the tree.

There are two mechanisms provided for enumerating all elements in an rbtree. The first, *rb_walk()*, is a simple callback implementation where the caller specifies a C function pointer and an untyped argument to be passed to it, and the tree code calls that function for each node in order. This has the advantage of a very simple implementation, at the cost of a somewhat more cumbersome API for the user (not unlike ISO C's `bsearch()` routine). It is a recursive implementation, however, and is thus not always available in environments that forbid the use of unbounded stack techniques like recursion.

There is also a *RB_FOR_EACH* iterator provided, which, like the similar APIs for the lists, works to iterate over a list in a more natural way, using a nested code block instead of a callback. It is also nonrecursive, though it requires log-sized space on the stack by default (however, this can be configured to use a fixed/maximally size buffer instead where needed to avoid the dynamic allocation). As with the lists, this is also available in a *RB_FOR_EACH_CONTAINER* variant which enumerates using a pointer to a container field and not the raw node pointer.

Tree Internals

As described, the Zephyr rbtree implementation is a conventional red/black tree as described pervasively in academic sources. Low level details about the algorithm are out of scope for this document, as they match existing conventions. This discussion will be limited to details notable or specific to the Zephyr implementation.

The core invariant guaranteed by the tree is that the path from the root of the tree to any leaf is no more than twice as long as the path to any other leaf. This is achieved by associating one bit of "color" with each node, either red or black, and enforcing a rule that no red child can be a child of another red child (i.e. that the number of black nodes on any path to the root must be the same, and that no more than that number of "extra" red nodes may be present). This rule is enforced by a set of rotation rules used to "fix" trees following modification.

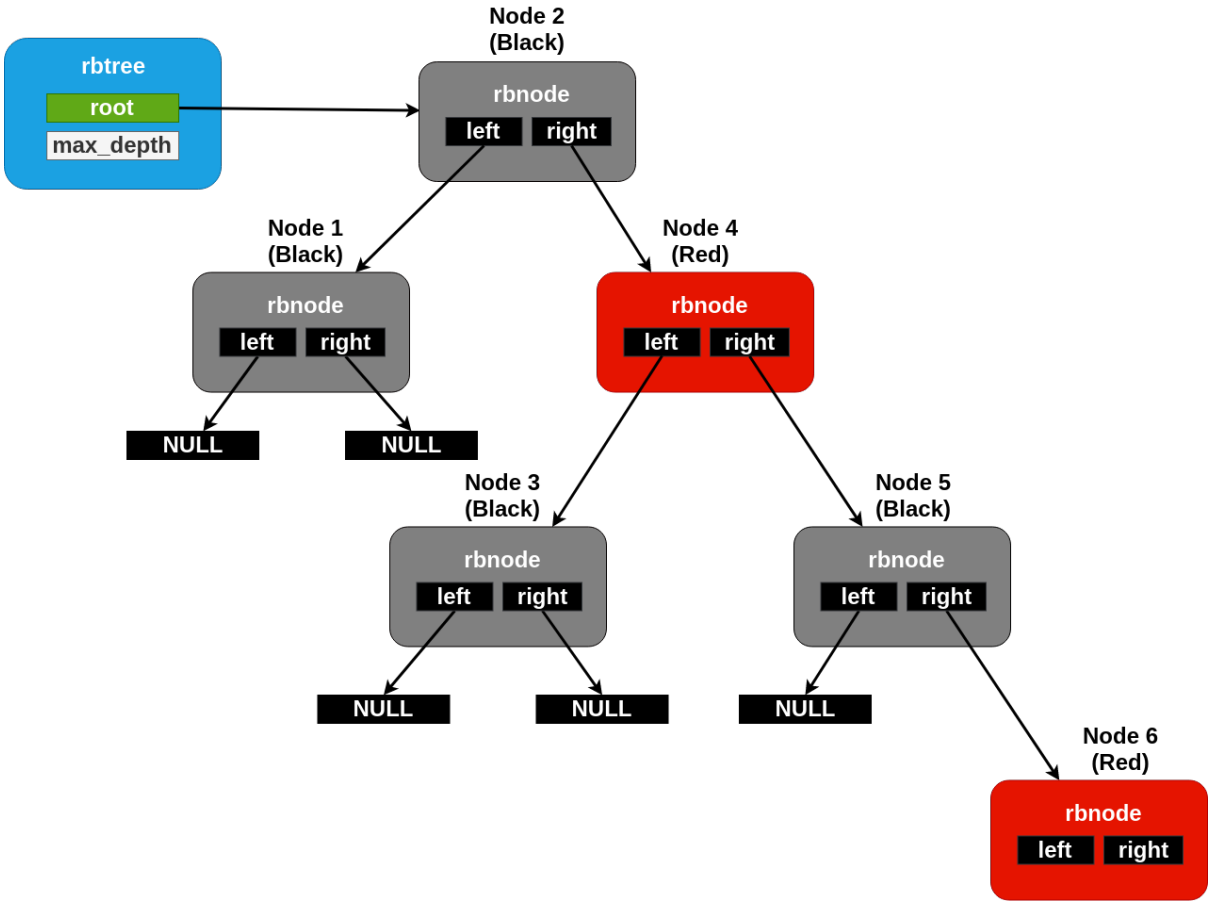


Fig. 8: A maximally unbalanced rbtree with a black height of two. No more nodes can be added underneath the rightmost node without rebalancing.

These rotations are conceptually implemented on top of a primitive that "swaps" the position of one node with another in the list. Typical implementations effect this by simply swapping the nodes internal "data" pointers, but because the Zephyr `rbnode` is intrusive, that cannot work. Zephyr must include somewhat more elaborate code to handle the edge cases (for example, one swapped node can be the root, or the two may already be parent/child).

The `rbnode` struct for a Zephyr rbtree contains only two pointers, representing the "left", and "right" children of a node within the binary tree. Traversal of a tree for rebalancing following modification, however, routinely requires the ability to iterate "upwards" from a node as well. It is very common for red/black trees in the industry to store a third "parent" pointer for this purpose. Zephyr avoids this requirement by building a "stack" of node pointers locally as it traverses downward thorugh the tree and updating it appropriately as modifications are made. So a Zephyr rbtree can be implemented with no more runtime storage overhead than a dlist.

These properties, of a balanced tree data structure that works with only two pointers of data per node and that works without any need for a memory allocation API, are quite rare in the industry and are somewhat unique to Zephyr.

Red/Black Tree API Reference

group `rbtree_apis`

Defines

`RB_FOR_EACH(tree, node)`

Walk a tree in-order without recursing.

While *rb_walk()* is very simple, recursing on the C stack can be clumsy for some purposes and on some architectures wastes significant memory in stack frames. This macro implements a non-recursive "foreach" loop that can iterate directly on the tree, at a moderate cost in code size.

Note that the resulting loop is not safe against modifications to the tree. Changes to the tree structure during the loop will produce incorrect results, as nodes may be skipped or duplicated. Unlike linked lists, no _SAFE variant exists.

Note also that the macro expands its arguments multiple times, so they should not be expressions with side effects.

Parameters

- `tree` – A pointer to a struct rbtree to walk
- `node` – The symbol name of a local struct rbnode* variable to use as the iterator

`RB_FOR_EACH_CONTAINER(tree, node, field)`

Loop over rbtree with implicit container field logic.

As for *RB_FOR_EACH()*, but "node" can have an arbitrary type containing a struct rbnode.

Parameters

- `tree` – A pointer to a struct rbtree to walk
- `node` – The symbol name of a local iterator
- `field` – The field name of a struct rbnode inside node

Typedefs

typedef bool (*`rb_lessthan_t`)(struct rbnode *a, struct rbnode *b)

Red/black tree comparison predicate.

Compares the two nodes and returns true if node A is strictly less than B according to the tree's sorting criteria, false otherwise.

Note that during insert, the new node being inserted will always be "A", where "B" is the existing node within the tree against which it is being compared. This trait can be used (with care!) to implement "most/least recently added" semantics between nodes which would otherwise compare as equal.

typedef void (*`rb_visit_t`)(struct rbnode *node, void *cookie)

Functions

void `rb_insert`(struct *rbtree* *tree, struct rbnode *node)

Insert node into tree.

void `rb_remove`(struct *rbtree* *tree, struct rbnode *node)

Remove node from tree.

static inline struct rbnode *`rb_get_min`(struct *rbtree* *tree)

Returns the lowest-sorted member of the tree.

static inline struct rbnode *`rb_get_max`(struct *rbtree* *tree)

Returns the highest-sorted member of the tree.

bool `rb_contains`(struct *rbtree* *tree, struct rbnode *node)

Returns true if the given node is part of the tree.

Note that this does not internally dereference the node pointer (though the tree's lessthan callback might!), it just tests it for equality with items in the tree. So it's feasible to use this to implement a "set" construct by simply testing the pointer value itself.

static inline void `rb_walk`(struct *rbtree* *tree, *rb_visit_t* visit_fn, void *cookie)

Walk/enumerate a rbtree.

Very simple recursive enumeration. Low code size, but requiring a separate function can be clumsy for the user and there is no way to break out of the loop early. See RB_FOR_EACH for an iterative implementation.

struct `rbtree`

#include <rb.h>

7.18.5 Ring Buffers

A *ring buffer* is a circular buffer, whose contents are stored in first-in-first-out order.

For circumstances where an application needs to implement asynchronous "streaming" copying of data, Zephyr provides a `struct ring_buf` abstraction to manage copies of such data in and out of a shared buffer of memory.

Two content data modes are supported:

- **Data item mode**: Multiple 32-bit word data items with metadata can be enqueued and dequeued from the ring buffer in chunks of up to 1020 bytes. Each data item also has two associated metadata values: a type identifier and a 16-bit integer value, both of which are application-specific.
- **Byte mode**: raw bytes can be enqueued and dequeued.

While the underlying data structure is the same, it is not legal to mix these two modes on a single ring buffer instance. A ring buffer initialized with a byte count must be used only with the "bytes" API, one initialized with a word count must use the "items" calls.

Concepts

Any number of ring buffers can be defined (limited only by available RAM). Each ring buffer is referenced by its memory address.

A ring buffer has the following key properties:

- A **data buffer** of 32-bit words or bytes. The data buffer contains the data items or raw bytes that have been added to the ring buffer but not yet removed.
- A **data buffer size**, measured in 32-bit words or bytes. This governs the maximum amount of data (including metadata values) the ring buffer can hold.

A ring buffer must be initialized before it can be used. This sets its data buffer to empty.

A `struct ring_buf` may be placed anywhere in user-accessible memory, and must be initialized with *ring_buf_init()* before use. This must be provided a region of user-controlled memory for use as the buffer itself. Note carefully that the units of the size of the buffer passed change (either bytes or words) depending on how the ring buffer will be used later. Macros for combining these steps in a single static declaration exist for convenience. *RING_BUF_DECLARE* will declare and statically initialize a ring buffer with a specified byte count, where *RING_BUF_ITEM_DECLARE_SIZE* will declare and statically initialize a buffer with a given count of 32 bit words. *RING_BUF_ITEM_DECLARE_POW2* can be used to initialize an items-mode buffer with a memory region guaranteed to be a power of two, which enables various optimizations internal to the implementation. No power-of-two initialization is available for bytes-mode ring buffers.

"Bytes" data may be copied into the ring buffer using *ring_buf_put()*, passing a data pointer and byte count. These bytes will be copied into the buffer in order, as many as will fit in the allocated buffer. The total number of bytes copied (which may be fewer than provided) will be returned. Likewise *ring_buf_get()* will copy bytes out of the ring buffer in the order that they were written, into a user-provided buffer, returning the number of bytes that were transferred.

To avoid multiply-copied-data situations, a "claim" API exists for byte mode. *ring_buf_put_claim()* takes a byte size value from the user and returns a pointer to memory internal to the ring buffer that can be used to receive those bytes, along with a size of the contiguous internal region (which may be smaller than requested). The user can then copy data into that region at a later time without assembling all the bytes in a single region first. When complete, *ring_buf_put_finish()* can be used to signal the buffer that the transfer is complete, passing the number of bytes actually transferred. At this point a new transfer can be initiated. Similarly, *ring_buf_get_claim()* returns a pointer to internal ring buffer data from which the user can read without making a verbatim copy, and *ring_buf_get_finish()* signals the buffer with how many bytes have been consumed and allows for a new transfer to begin.

"Items" mode works similarly to bytes mode, except that all transfers are in units of 32 bit words and all memory is assumed to be aligned on 32 bit boundaries. The write and read operations are *ring_buf_item_put()* and *ring_buf_item_get()*, and work otherwise identically to the bytes mode APIs. There no "claim" API provided for items mode. One important difference is that unlike *ring_buf_put()*, *ring_buf_item_put()* will not do a partial transfer; it will return an error in the case where the provided data does not fit in its entirety.

The user can manage the capacity of a ring buffer without modifying it using the *ring_buf_space_get()* call (which returns a value of either bytes or items depending on how the ring buffer has been used), or by testing the *ring_buf_is_empty()* predicate.

Finally, a *ring_buf_reset()* call exists to immediately empty a ring buffer, discarding the tracking of any bytes or items already written to the buffer. It does not modify the memory contents of the buffer itself, however.

Data item mode A **data item mode** ring buffer instance is declared using *RING_BUF_ITEM_DECLARE_POW2()* or *RING_BUF_ITEM_DECLARE_SIZE()* and accessed using *ring_buf_item_put()* and *ring_buf_item_get()*.

A ring buffer **data item** is an array of 32-bit words from 0 to 1020 bytes in length. When a data item is **enqueued** (*ring_buf_item_put()*) its contents are copied to the data buffer, along with its associated metadata values (which occupy one additional 32-bit word). If the ring buffer has insufficient space to hold the new data item the enqueue operation fails.

A data items is **dequeued** (*ring_buf_item_get()*) from a ring buffer by removing the oldest enqueued item. The contents of the dequeued data item, as well as its two metadata values, are copied to areas supplied by the retriever. If the ring buffer is empty, or if the data array supplied by the retriever is not large enough to hold the data item's data, the dequeue operation fails.

Byte mode A **byte mode** ring buffer instance is declared using *RING_BUF_ITEM_DECLARE_SIZE()* and accessed using: *ring_buf_put_claim()*, *ring_buf_put_finish()*, *ring_buf_get_claim()*, *ring_buf_get_finish()*, *ring_buf_put()* and *ring_buf_get()*.

Data can be copied into the ring buffer (see *ring_buf_put()*) or ring buffer memory can be used directly by the user. In the latter case, the operation is split into three stages:

1. allocating the buffer (*ring_buf_put_claim()*) when user requests the destination location where data can be written.
2. writing the data by the user (e.g. buffer written by DMA).
3. indicating the amount of data written to the provided buffer (*ring_buf_put_finish()*). The amount can be less than or equal to the allocated amount.

Data can be retrieved from a ring buffer through copying (see *ring_buf_get()*) or accessed directly by address. In the latter case, the operation is split into three stages:

1. retrieving source location with valid data written to a ring buffer (see *ring_buf_get_claim()*).
2. processing data
3. freeing processed data (see *ring_buf_get_finish()*). The amount freed can be less than or equal or to the retrieved amount.

Concurrency The ring buffer APIs do not provide any concurrency control. Depending on usage (particularly with respect to number of concurrent readers/writers) applications may need to protect the ring buffer with mutexes and/or use semaphores to notify consumers that there is data to read.

For the trivial case of one producer and one consumer, concurrency shouldn't be needed.

Internal Operation If the size of the data buffer is a power of two, the ring buffer uses efficient masking operations instead of expensive modulo operations when enqueuing and dequeuing data items. This option is applicable only for data item mode.

Data streamed through a ring buffer is always written to the next byte within the buffer, wrapping around to the first element after reaching the end, thus the "ring" structure. Internally, the `struct ring_buf` contains its own buffer pointer and its size, and also a "head" and "tail" index representing where the next read and write

This boundary is invisible to the user using the normal put/get APIs, but becomes a barrier to the "claim" API, because obviously no contiguous region can be returned that crosses the end of the buffer. This can

be surprising to application code, and produce performance artifacts when transfers need to alias closely to the size of the buffer, as the number of calls to claim/finish need to double for such transfers.

When running in items mode (only), the ring buffer contains two implementations for the modular arithmetic required to compute "next element" offsets. One is used for arbitrary sized buffers, but the other is optimized for power of two sizes and can replace the compare and subtract steps with a simple bitmask in several places, at the cost of testing the "mask" value for each call.

Implementation

Defining a Ring Buffer A ring buffer is defined using a variable of type `ring_buf`. It must then be initialized by calling *ring_buf_init()*.

The following code defines and initializes an empty **data item mode** ring buffer (which is part of a larger data structure). The ring buffer's data buffer is capable of holding 64 words of data and metadata information.

```
#define MY_RING_BUF_SIZE 64

struct my_struct {
    struct ring_buf rb;
    uint32_t buffer[MY_RING_BUF_SIZE];
    ...
};
struct my_struct ms;

void init_my_struct {
    ring_buf_init(&ms.rb, sizeof(ms.buffer), ms.buffer);
    ...
}
```

Alternatively, a ring buffer can be defined and initialized at compile time using one of two macros at file scope. Each macro defines both the ring buffer itself and its data buffer.

The following code defines a ring buffer with a power-of-two sized data buffer, which can be accessed using efficient masking operations.

```
/* Buffer with 2^8 (or 256) words */
RING_BUF_ITEM_DECLARE_POW2(my_ring_buf, 8);
```

The following code defines an application-specific sized **byte mode** ring buffer enqueued and dequeued as raw bytes:

```
#define MY_RING_BUF_WORDS 93
RING_BUF_ITEM_DECLARE_SIZE(my_ring_buf, MY_RING_BUF_WORDS);
```

The following code defines a ring buffer with an arbitrary-sized data buffer, which can be accessed using less efficient modulo operations. Ring buffer is intended to be used for raw bytes.

```
#define MY_RING_BUF_BYTES 93
RING_BUF_DECLARE_SIZE(my_ring_buf, MY_RING_BUF_BYTES);
```

Enqueuing Data A data item is added to a ring buffer by calling *ring_buf_item_put()*.

```
uint32_t data[MY_DATA_WORDS];
int ret;

ret = ring_buf_item_put(&ring_buf, TYPE_FOO, 0, data, SIZE32_OF(data));
```

(continues on next page)

(continued from previous page)

```
if (ret == -EMSGSIZE) {
    /* not enough room for the data item */
    ...
}
```

If the data item requires only the type or application-specific integer value (i.e. it has no data array), a size of 0 and data pointer of `NULL` can be specified.

```
int ret;

ret = ring_buf_item_put(&ring_buf, TYPE_BAR, 17, NULL, 0);
if (ret == -EMSGSIZE) {
    /* not enough room for the data item */
    ...
}
```

Bytes are copied to a **byte mode** ring buffer by calling *ring_buf_put()*.

```
uint8_t my_data[MY_RING_BUF_BYTES];
uint32_t ret;

ret = ring_buf_put(&ring_buf, my_data, SIZE_OF(my_data));
if (ret != SIZE_OF(my_data)) {
    /* not enough room, partial copy. */
    ...
}
```

Data can be added to a **byte mode** ring buffer by directly accessing the ring buffer's memory. For example:

```
uint32_t size;
uint32_t rx_size;
uint8_t *data;
int err;

/* Allocate buffer within a ring buffer memory. */
size = ring_buf_put_claim(&ring_buf, &data, MY_RING_BUF_BYTES);

/* Work directly on a ring buffer memory. */
rx_size = uart_rx(data, size);

/* Indicate amount of valid data. rx_size can be equal or less than size. */
err = ring_buf_put_finish(&ring_buf, rx_size);
if (err != 0) {
    /* No space to put requested amount of data to ring buffer. */
    ...
}
```

Retrieving Data A data item is removed from a ring buffer by calling *ring_buf_item_get()*.

```
uint32_t my_data[MY_DATA_WORDS];
uint16_t my_type;
uint8_t  my_value;
uint8_t  my_size;
int ret;
```

(continues on next page)

(continued from previous page)

```
my_size = SIZE32_OF(my_data);
ret = ring_buf_item_get(&ring_buf, &my_type, &my_value, my_data, &my_size);
if (ret == -EMSGSIZE) {
    printk("Buffer is too small, need %d uint32_t\n", my_size);
} else if (ret == -EAGAIN) {
    printk("Ring buffer is empty\n");
} else {
    printk("Got item of type %u value &u of size %u dwords\n",
           my_type, my_value, my_size);
    ...
}
```

Data bytes are copied out from a **byte mode** ring buffer by calling *ring_buf_get()*. For example:

```
uint8_t my_data[MY_DATA_BYTES];
size_t  ret;

ret = ring_buf_get(&ring_buf, my_data, sizeof(my_data));
if (ret != sizeof(my_size)) {
    /* Less bytes copied. */
} else {
    /* Requested amount of bytes retrieved. */
    ...
}
```

Data can be retrieved from a **byte mode** ring buffer by direct operations on the ring buffer's memory. For example:

```
uint32_t size;
uint32_t proc_size;
uint8_t *data;
int err;

/* Get buffer within a ring buffer memory. */
size = ring_buf_get_claim(&ring_buf, &data, MY_RING_BUF_BYTES);

/* Work directly on a ring buffer memory. */
proc_size = process(data, size);

/* Indicate amount of data that can be freed. proc_size can be equal or less
 * than size.
 */
err = ring_buf_get_finish(&ring_buf, proc_size);
if (err != 0) {
    /* proc_size exceeds amount of valid data in a ring buffer. */
    ...
}
```

Configuration Options

Related configuration options:

- `CONFIG_RING_BUFFER`: Enable ring buffer.

API Reference

The following ring buffer APIs are provided by include/sys/ring_buffer.h:

group `ring_buffer_apis`

Defines

`RING_BUF_ITEM_DECLARE_POW2(name, pow)`

Define and initialize a high performance ring buffer.

This macro establishes a ring buffer whose size must be a power of 2; that is, the ring buffer contains 2 ^ pow 32-bit words, where *pow* is the specified ring buffer size exponent. A high performance ring buffer doesn't require the use of modulo arithmetic operations to maintain itself.

The ring buffer can be accessed outside the module where it is defined using:

```
extern struct ring_buf <name>;
```

Parameters

- `name` – Name of the ring buffer.
- `pow` – Ring buffer size exponent.

`RING_BUF_ITEM_DECLARE_SIZE(name, size32)`

Define and initialize a standard ring buffer.

This macro establishes a ring buffer of an arbitrary size. A standard ring buffer uses modulo arithmetic operations to maintain itself.

The ring buffer can be accessed outside the module where it is defined using:

```
extern struct ring_buf <name>;
```

Parameters

- `name` – Name of the ring buffer.
- `size32` – Size of ring buffer (in 32-bit words).

`RING_BUF_DECLARE(name, size8)`

Define and initialize a ring buffer for byte data.

This macro establishes a ring buffer of an arbitrary size.

The ring buffer can be accessed outside the module where it is defined using:

```
extern struct ring_buf <name>;
```

Parameters

- `name` – Name of the ring buffer.
- `size8` – Size of ring buffer (in bytes).

Functions

static inline void `ring_buf_init`(struct ring_buf *buf, uint32_t size, void *data)

Initialize a ring buffer.

This routine initializes a ring buffer, prior to its first use. It is only used for ring buffers not defined using RING_BUF_DECLARE, RING_BUF_ITEM_DECLARE_POW2 or RING_BUF_ITEM_DECLARE_SIZE.

Setting *size* to a power of 2 establishes a high performance ring buffer that doesn't require the use of modulo arithmetic operations to maintain itself.

Parameters

- `buf` – Address of ring buffer.
- `size` – Ring buffer size (in 32-bit words or bytes).
- `data` – Ring buffer data area (uint32_t data[size] or uint8_t data[size] for bytes mode).

int `ring_buf_is_empty`(struct ring_buf *buf)

Determine if a ring buffer is empty.

Parameters

- `buf` – Address of ring buffer.

Returns 1 if the ring buffer is empty, or 0 if not.

static inline void `ring_buf_reset`(struct ring_buf *buf)

Reset ring buffer state.

Parameters

- `buf` – Address of ring buffer.

uint32_t `ring_buf_space_get`(struct ring_buf *buf)

Determine free space in a ring buffer.

Parameters

- `buf` – Address of ring buffer.

Returns Ring buffer free space (in 32-bit words or bytes).

static inline uint32_t `ring_buf_capacity_get`(struct ring_buf *buf)

Return ring buffer capacity.

Parameters

- `buf` – Address of ring buffer.

Returns Ring buffer capacity (in 32-bit words or bytes).

uint32_t `ring_buf_size_get`(struct ring_buf *buf)

Determine used space in a ring buffer.

Parameters

- `buf` – Address of ring buffer.

Returns Ring buffer space used (in 32-bit words or bytes).

int `ring_buf_item_put`(struct ring_buf *buf, uint16_t type, uint8_t value, uint32_t *data, uint8_t size32)

Write a data item to a ring buffer.

This routine writes a data item to ring buffer *buf*. The data item is an array of 32-bit words (from zero to 1020 bytes in length), coupled with a 16-bit type identifier and an 8-bit integer value.

Warning: Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Parameters

- `buf` – Address of ring buffer.
- `type` – Data item's type identifier (application specific).
- `value` – Data item's integer value (application specific).
- `data` – Address of data item.
- `size32` – Data item size (number of 32-bit words).

Return values

- `0` – Data item was written.
- `-EMSGSIZE` – Ring buffer has insufficient free space.

int `ring_buf_item_get`(struct ring_buf *buf, uint16_t *type, uint8_t *value, uint32_t *data, uint8_t *size32)

Read a data item from a ring buffer.

This routine reads a data item from ring buffer *buf*. The data item is an array of 32-bit words (up to 1020 bytes in length), coupled with a 16-bit type identifier and an 8-bit integer value.

Warning: Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Parameters

- `buf` – Address of ring buffer.
- `type` – Area to store the data item's type identifier.
- `value` – Area to store the data item's integer value.
- `data` – Area to store the data item. Can be NULL to discard data.
- `size32` – Size of the data item storage area (number of 32-bit chunks).

Return values

- `0` – Data item was fetched; *size32* now contains the number of 32-bit words read into data area *data*.
- `-EAGAIN` – Ring buffer is empty.
- `-EMSGSIZE` – Data area *data* is too small; *size32* now contains the number of 32-bit words needed.

uint32_t `ring_buf_put_claim`(struct ring_buf *buf, uint8_t **data, uint32_t size)

Allocate buffer for writing data to a ring buffer.

With this routine, memory copying can be reduced since internal ring buffer can be used directly by the user. Once data is written to allocated area number of bytes written can be confirmed (see *ring_buf_put_finish*).

Warning: Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Warning: Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- `buf` – **[in]** Address of ring buffer.
- `data` – **[out]** Pointer to the address. It is set to a location within ring buffer.
- `size` – **[in]** Requested allocation size (in bytes).

Returns Size of allocated buffer which can be smaller than requested if there is not enough free space or buffer wraps.

int `ring_buf_put_finish`(struct ring_buf *buf, uint32_t size)

Indicate number of bytes written to allocated buffers.

Warning: Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Warning: Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `size` – Number of valid bytes in the allocated buffers.

Return values

- `0` – Successful operation.
- `-EINVAL` – Provided *size* exceeds free space in the ring buffer.

uint32_t `ring_buf_put`(struct ring_buf *buf, const uint8_t *data, uint32_t size)

Write (copy) data to a ring buffer.

This routine writes data to a ring buffer *buf*.

Warning: Use cases involving multiple writers to the ring buffer must prevent concurrent write operations, either by preventing all writers from being preempted or by using a mutex to govern writes to the ring buffer.

Warning: Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `data` – Address of data.
- `size` – Data size (in bytes).

Return values `Number` – of bytes written.

uint32_t `ring_buf_get_claim`(struct ring_buf *buf, uint8_t **data, uint32_t size)

Get address of a valid data in a ring buffer.

With this routine, memory copying can be reduced since internal ring buffer can be used directly by the user. Once data is processed it can be freed using *ring_buf_get_finish*.

Warning: Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning: Ring buffer instance should not mix byte access and item access (calls prefixed with ring_buf_item_).

Parameters

- `buf` – **[in]** Address of ring buffer.
- `data` – **[out]** Pointer to the address. It is set to a location within ring buffer.
- `size` – **[in]** Requested size (in bytes).

Returns Number of valid bytes in the provided buffer which can be smaller than requested if there is not enough free space or buffer wraps.

int `ring_buf_get_finish`(struct ring_buf *buf, uint32_t size)

Indicate number of bytes read from claimed buffer.

Warning: Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning: Ring buffer instance should not mix byte access and item mode (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `size` – Number of bytes that can be freed.

Return values

- `0` – Successful operation.

- `-EINVAL` – Provided *size* exceeds valid bytes in the ring buffer.

uint32_t `ring_buf_get`(struct ring_buf *buf, uint8_t *data, uint32_t size)

Read data from a ring buffer.

This routine reads data from a ring buffer *buf*.

Warning: Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning: Ring buffer instance should not mix byte access and item mode (calls prefixed with ring_buf_item_).

Parameters

- `buf` – Address of ring buffer.
- `data` – Address of the output buffer. Can be NULL to discard data.
- `size` – Data size (in bytes).

Return values `Number` – of bytes written to the output buffer.

uint32_t `ring_buf_peek`(struct ring_buf *buf, uint8_t *data, uint32_t size)

Peek at data from a ring buffer.

This routine reads data from a ring buffer *buf* without removal.

Warning: Use cases involving multiple reads of the ring buffer must prevent concurrent read operations, either by preventing all readers from being preempted or by using a mutex to govern reads to the ring buffer.

Warning: Ring buffer instance should not mix byte access and item mode (calls prefixed with ring_buf_item_).

Warning: Multiple calls to peek will result in the same data being 'peeked' multiple times. To remove data, use either *ring_buf_get* or *ring_buf_get_claim* followed by *ring_buf_get_finish* with a non-zero `size`.

Parameters

- `buf` – Address of ring buffer.
- `data` – Address of the output buffer. Cannot be NULL.
- `size` – Data size (in bytes).

Return values `Number` – of bytes written to the output buffer.

7.19 MODBUS

Modbus is an industrial messaging protocol. The protocol is specified for different types of networks or buses. Zephyr OS implementation supports communication over serial line and may be used with different physical interfaces, like RS485 or RS232. TCP support is not implemented directly, but there are helper functions to realize TCP support according to the application's needs.

Modbus communication is based on client/server model. Only one client may be present on the bus. Client can communicate with several server devices. Server devices themselves are passive and must not send requests or unsolicited responses. Services requested by the client are specified by function codes (FCxx), and can be found in the specification or documentation of the API below.

Zephyr RTOS implementation supports both client and server roles.

More information about Modbus and Modbus RTU can be found on the website MODBUS Protocol Specifications.

7.19.1 Samples

`modbus-rtu-server-sample` and `modbus-rtu-client-sample` give the possibility to try out RTU server and RTU client implementation with an evaluation board.

`modbus-tcp-server-sample` is a simple Modbus TCP server.

`modbus-gateway-sample` is an example how to build a TCP to serial line gateway with Zephyr OS.

7.19.2 API Reference

group `modbus`

MODBUS transport protocol API.

Defines

`MODBUS_MBAP_LENGTH`

Length of MBAP Header

`MODBUS_MBAP_AND_FC_LENGTH`

Length of MBAP Header plus function code

Typedefs

typedef int (*`modbus_raw_cb_t`)(const int iface, const struct *modbus_adu* *adu)

ADU raw callback function signature.

Param iface Modbus RTU interface index

Param adu Pointer to the RAW ADU struct to send

Retval 0 If transfer was successful

Enums

enum `modbus_mode`

Modbus interface mode.

Values:

enumerator `MODBUS_MODE_RTU`

Modbus over serial line RTU mode

enumerator `MODBUS_MODE_ASCII`

Modbus over serial line ASCII mode

enumerator `MODBUS_MODE_RAW`

Modbus raw ADU mode

Functions

int `modbus_read_coils`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint8_t *const coil_tbl, const uint16_t num_coils)

Coil read (FC01)

Sends a Modbus message to read the status of coils from a server.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Coil starting address
- `coil_tbl` – Pointer to an array of bytes containing the value of the coils read. The format is:

```
                MSB                                 LSB
                B7   B6   B5   B4   B3   B2   B1   B0
                -------------------------------------
coil_tbl[0]     #8   #7                          #1
coil_tbl[1]     #16  #15                         #9
     :
     :
```

  Note that the array that will be receiving the coil values must be greater than or equal to: (num_coils - 1) / 8 + 1

- `num_coils` – Quantity of coils to read

Return values `0` – If the function was successful

int `modbus_read_dinputs`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint8_t *const di_tbl, const uint16_t num_di)

Read discrete inputs (FC02)

Sends a Modbus message to read the status of discrete inputs from a server.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server

- `start_addr` – Discrete input starting address
- `di_tbl` – Pointer to an array that will receive the state of the discrete inputs. The format of the array is as follows:

```
              MSB                                   LSB
              B7   B6   B5   B4   B3   B2   B1   B0
              ---------------------------------------
di_tbl[0]     #8   #7                             #1
di_tbl[1]     #16  #15                            #9
     :
     :
```

  Note that the array that will be receiving the discrete input values must be greater than or equal to: (num_di - 1) / 8 + 1

- `num_di` – Quantity of discrete inputs to read

Return values `0` – If the function was successful

int `modbus_read_holding_regs`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint16_t *const reg_buf, const uint16_t num_regs)

Read holding registers (FC03)

Sends a Modbus message to read the value of holding registers from a server.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array that will receive the current values of the holding registers from the server. The array pointed to by 'reg_buf' needs to be able to hold at least 'num_regs' entries.
- `num_regs` – Quantity of registers to read

Return values `0` – If the function was successful

int `modbus_read_input_regs`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint16_t *const reg_buf, const uint16_t num_regs)

Read input registers (FC04)

Sends a Modbus message to read the value of input registers from a server.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array that will receive the current value of the holding registers from the server. The array pointed to by 'reg_buf' needs to be able to hold at least 'num_regs' entries.
- `num_regs` – Quantity of registers to read

Return values `0` – If the function was successful

int `modbus_write_coil`(const int iface, const uint8_t unit_id, const uint16_t coil_addr, const bool coil_state)

Write single coil (FC05)

Sends a Modbus message to write the value of single coil to a server.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `coil_addr` – Coils starting address
- `coil_state` – Is the desired state of the coil

Return values `0` – If the function was successful

int `modbus_write_holding_reg`(const int iface, const uint8_t unit_id, const uint16_t start_addr, const uint16_t reg_val)

Write single holding register (FC06)

Sends a Modbus message to write the value of single holding register to a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Coils starting address
- `reg_val` – Desired value of the holding register

Return values `0` – If the function was successful

int `modbus_request_diagnostic`(const int iface, const uint8_t unit_id, const uint16_t sfunc, const uint16_t data, uint16_t *const data_out)

Read diagnostic (FC08)

Sends a Modbus message to perform a diagnostic function of a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `sfunc` – Diagnostic sub-function code
- `data` – Sub-function data
- `data_out` – Pointer to the data value

Return values `0` – If the function was successful

int `modbus_write_coils`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint8_t *const coil_tbl, const uint16_t num_coils)

Write coils (FC15)

Sends a Modbus message to write to coils on a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Coils starting address
- `coil_tbl` – Pointer to an array of bytes containing the value of the coils to write. The format is:

```
                MSB                                 LSB
                B7   B6   B5   B4   B3   B2   B1   B0
                ---------------------------------------
coil_tbl[0]     #8   #7                          #1
```

(continues on next page)

(continued from previous page)

```
coil_tbl[1]      #16  #15                                  #9
     :
     :
```

Note that the array that will be receiving the coil values must be greater than or equal to: (num_coils - 1) / 8 + 1

- `num_coils` – Quantity of coils to write

Return values `0` – If the function was successful

int `modbus_write_holding_regs`(const int iface, const uint8_t unit_id, const uint16_t start_addr, uint16_t *const reg_buf, const uint16_t num_regs)

Write holding registers (FC16)

Sends a Modbus message to write to integer holding registers to a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array containing the value of the holding registers to write. Note that the array containing the register values must be greater than or equal to 'num_regs'
- `num_regs` – Quantity of registers to write

Return values `0` – If the function was successful

int `modbus_read_holding_regs_fp`(const int iface, const uint8_t unit_id, const uint16_t start_addr, float *const reg_buf, const uint16_t num_regs)

Read floating-point holding registers (FC03)

Sends a Modbus message to read the value of floating-point holding registers from a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address
- `reg_buf` – Is a pointer to an array that will receive the current values of the holding registers from the server. The array pointed to by 'reg_buf' needs to be able to hold at least 'num_regs' entries.
- `num_regs` – Quantity of registers to read

Return values `0` – If the function was successful

int `modbus_write_holding_regs_fp`(const int iface, const uint8_t unit_id, const uint16_t start_addr, float *const reg_buf, const uint16_t num_regs)

Write floating-point holding registers (FC16)

Sends a Modbus message to write to floating-point holding registers to a server unit.

Parameters

- `iface` – Modbus interface index
- `unit_id` – Modbus unit ID of the server
- `start_addr` – Register starting address

- `reg_buf` – Is a pointer to an array containing the value of the holding registers to write. Note that the array containing the register values must be greater than or equal to 'num_regs'
- `num_regs` – Quantity of registers to write

Return values `0` – If the function was successful

int `modbus_iface_get_by_name`(const char *iface_name)

Get Modbus interface index according to interface name.

If there is more than one interface, it can be used to clearly identify interfaces in the application.

Parameters

- `iface_name` – Modbus interface name

Return values `Modbus` – interface index or negative error value.

int `modbus_init_server`(const int iface, struct *modbus_iface_param* param)

Configure Modbus Interface as raw ADU server.

Parameters

- `iface` – Modbus RTU interface index
- `param` – Configuration parameter of the server interface

Return values `0` – If the function was successful

int `modbus_init_client`(const int iface, struct *modbus_iface_param* param)

Configure Modbus Interface as raw ADU client.

Parameters

- `iface` – Modbus RTU interface index
- `param` – Configuration parameter of the client interface

Return values `0` – If the function was successful

int `modbus_disable`(const uint8_t iface)

Disable Modbus Interface.

This function is called to disable Modbus interface.

Parameters

- `iface` – Modbus interface index

Return values `0` – If the function was successful

int `modbus_raw_submit_rx`(const int iface, const struct *modbus_adu* *adu)

Submit raw ADU.

Parameters

- `iface` – Modbus RTU interface index
- `adu` – Pointer to the RAW ADU struct that is received

Return values `0` – If transfer was successful

void `modbus_raw_put_header`(const struct *modbus_adu* *adu, uint8_t *header)

Put MBAP header into a buffer.

Parameters

- `adu` – Pointer to the RAW ADU struct
- `header` – Pointer to the buffer in which MBAP header will be placed.

Return values `0` – If transfer was successful

void `modbus_raw_get_header`(struct *modbus_adu* *adu, const uint8_t *header)

Get MBAP header from a buffer.

Parameters

- `adu` – Pointer to the RAW ADU struct
- `header` – Pointer to the buffer containing MBAP header

Return values `0` – If transfer was successful

void `modbus_raw_set_server_failure`(struct *modbus_adu* *adu)

Set Server Device Failure exception.

This function modifies ADU passed by the pointer.

Parameters

- `adu` – Pointer to the RAW ADU struct

int `modbus_raw_backend_txn`(const int iface, struct *modbus_adu* *adu)

Use interface as backend to send and receive ADU.

This function overwrites ADU passed by the pointer and generates exception responses if backend interface is misconfigured or target device is unreachable.

Parameters

- `iface` – Modbus client interface index
- `adu` – Pointer to the RAW ADU struct

Return values `0` – If transfer was successful

struct `modbus_adu`

#include <modbus.h> Frame struct used internally and for raw ADU support.

Public Members

uint16_t `trans_id`

Transaction Identifier

uint16_t `proto_id`

Protocol Identifier

uint16_t `length`

Length of the data only (not the length of unit ID + PDU)

uint8_t `unit_id`

Unit Identifier

uint8_t `fc`

Function Code

uint8_t `data`[CONFIG_MODBUS_BUFFER_SIZE - 4]

Transaction Data

uint16_t `crc`

RTU CRC

struct `modbus_user_callbacks`

#include <modbus.h> Modbus Server User Callback structure

Public Members

int (*`coil_rd`)(uint16_t addr, bool *state)

Coil read callback

int (*`coil_wr`)(uint16_t addr, bool state)

Coil write callback

int (*`discrete_input_rd`)(uint16_t addr, bool *state)

Discrete Input read callback

int (*`input_reg_rd`)(uint16_t addr, uint16_t *reg)

Input Register read callback

int (*`input_reg_rd_fp`)(uint16_t addr, float *reg)

Floating Point Input Register read callback

int (*`holding_reg_rd`)(uint16_t addr, uint16_t *reg)

Holding Register read callback

int (*`holding_reg_wr`)(uint16_t addr, uint16_t reg)

Holding Register write callback

int (*`holding_reg_rd_fp`)(uint16_t addr, float *reg)

Floating Point Holding Register read callback

int (*`holding_reg_wr_fp`)(uint16_t addr, float reg)

Floating Point Holding Register write callback

struct `modbus_serial_param`

#include <modbus.h> Modbus serial line parameter.

Public Members

uint32_t `baud`

Baudrate of the serial line

enum *uart_config_parity* `parity`

parity UART's parity setting: UART_CFG_PARITY_NONE, UART_CFG_PARITY_EVEN, UART_CFG_PARITY_ODD

struct `modbus_server_param`

#include <modbus.h> Modbus server parameter.

Public Members

struct *modbus_user_callbacks* `*user_cb`

Pointer to the User Callback structure

uint8_t `unit_id`

Modbus unit ID of the server

struct `modbus_iface_param`

#include <modbus.h> User parameter structure to configure Modbus interfase as client or server.

Public Members

enum *modbus_mode* `mode`

Mode of the interface

uint32_t `rx_timeout`

Amount of time client will wait for a response from the server.

struct *modbus_serial_param* `serial`

Serial support parameter of the interface

modbus_raw_cb_t `raw_tx_cb`

Pointer to raw ADU callback function

7.20 Networking

7.20.1 Network APIs

BSD Sockets

Overview Zephyr offers an implementation of a subset of the BSD Sockets API (a part of the POSIX standard). This API allows to reuse existing programming experience and port existing simple networking applications to Zephyr.

Here are the key requirements and concepts which governed BSD Sockets compatible API implementation for Zephyr:

- Has minimal overhead, similar to the requirement for other Zephyr subsystems.
- Is namespaced by default, to avoid name conflicts with well-known names like `close()`, which may be part of libc or other POSIX compatibility libraries. If enabled by `CONFIG_NET_SOCKETS_POSIX_NAMES`, it will also expose native POSIX names.

BSD Sockets compatible API is enabled using `CONFIG_NET_SOCKETS` config option and implements the following operations: `socket()`, `close()`, `recv()`, `recvfrom()`, `send()`, `sendto()`, `connect()`, `bind()`, `listen()`, `accept()`, `fcntl()` (to set non-blocking mode), `getsockopt()`, `setsockopt()`, `poll()`, `select()`, `getaddrinfo()`, `getnameinfo()`.

Based on the namespacing requirements above, these operations are by default exposed as functions with `zsock_` prefix, e.g. *zsock_socket()* and *zsock_close()*. If the config option `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined, all the functions will be also exposed as aliases without the prefix. This includes the functions like `close()` and `fcntl()` (which may conflict with functions in libc or other libraries, for example, with the filesystem libraries).

Another entailment of the design requirements above is that the Zephyr API aggressively employs the short-read/short-write property of the POSIX API whenever possible (to minimize complexity and overheads). POSIX allows for calls like `recv()` and `send()` to actually process (receive or send) less data than requested by the user (on `SOCK_STREAM` type sockets). For example, a call `recv(sock, 1000, 0)` may return 100, meaning that only 100 bytes were read (short read), and the application needs to retry call(s) to receive the remaining 900 bytes.

The BSD Sockets API uses file descriptors to represent sockets. File descriptors are small integers, consecutively assigned from zero, shared among sockets, files, special devices (like stdin/stdout), etc. Internally, there is a table mapping file descriptors to internal object pointers. The file descriptor table is used by the BSD Sockets API even if the rest of the POSIX subsystem (filesystem, stdin/stdout) is not enabled.

Secure Sockets Zephyr provides an extension of standard POSIX socket API, allowing to create and configure sockets with TLS protocol types, facilitating secure communication. Secure functions for the implementation are provided by mbedTLS library. Secure sockets implementation allows use of both TLS and DTLS protocols with standard socket calls. See *net_ip_protocol_secure* type for supported secure protocol versions.

To enable secure sockets, set the `CONFIG_NET_SOCKETS_SOCKOPT_TLS` option. To enable DTLS support, use `CONFIG_NET_SOCKETS_ENABLE_DTLS` option.

TLS credentials subsystem TLS credentials must be registered in the system before they can be used with secure sockets. See *tls_credential_add()* for more information.

When a specific TLS credential is registered in the system, it is assigned with numeric value of type *sec_tag_t*, called a tag. This value can be used later on to reference the credential during secure socket configuration with socket options.

The following TLS credential types can be registered in the system:

- `TLS_CREDENTIAL_CA_CERTIFICATE`
- `TLS_CREDENTIAL_SERVER_CERTIFICATE`
- `TLS_CREDENTIAL_PRIVATE_KEY`
- `TLS_CREDENTIAL_PSK`
- `TLS_CREDENTIAL_PSK_ID`

An example registration of CA certificate (provided in `ca_certificate` array) looks like this:

```
ret = tls_credential_add(CA_CERTIFICATE_TAG, TLS_CREDENTIAL_CA_CERTIFICATE,
                         ca_certificate, sizeof(ca_certificate));
```

By default certificates in DER format are supported. PEM support can be enabled in mbedTLS settings.

Secure Socket Creation A secure socket can be created by specifying secure protocol type, for instance:

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TLS_1_2);
```

Once created, it can be configured with socket options. For instance, the CA certificate and hostname can be set:

```
sec_tag_t sec_tag_opt[] = {
        CA_CERTIFICATE_TAG,
};

ret = setsockopt(sock, SOL_TLS, TLS_SEC_TAG_LIST,
                 sec_tag_opt, sizeof(sec_tag_opt));
```

```
char host[] = "google.com";

ret = setsockopt(sock, SOL_TLS, TLS_HOSTNAME, host, sizeof(host) - 1);
```

Once configured, socket can be used just like a regular TCP socket.

Several samples in Zephyr use secure sockets for communication. For a sample use see e.g. echo-server sample application or HTTP GET sample application.

Secure Sockets options Secure sockets offer the following options for socket management:

group `secure_sockets_options`

Defines

`TLS_SEC_TAG_LIST`

Socket option to select TLS credentials to use. It accepts and returns an array of sec_tag_t that indicate which TLS credentials should be used with specific socket.

`TLS_HOSTNAME`

Write-only socket option to set hostname. It accepts a string containing the hostname (may be NULL to disable hostname verification). By default, hostname check is enforced for TLS clients.

`TLS_CIPHERSUITE_LIST`

Socket option to select ciphersuites to use. It accepts and returns an array of integers with IANA assigned ciphersuite identifiers. If not set, socket will allow all ciphersuites available in the system (mebdTLS default behavior).

`TLS_CIPHERSUITE_USED`

Read-only socket option to read a ciphersuite chosen during TLS handshake. It returns an integer containing an IANA assigned ciphersuite identifier of chosen ciphersuite.

TLS_PEER_VERIFY

Write-only socket option to set peer verification level for TLS connection. This option accepts an integer with a peer verification level, compatible with mbedTLS values:

- 0 - none
- 1 - optional
- 2 - required

If not set, socket will use mbedTLS defaults (none for servers, required for clients).

TLS_DTLS_ROLE

Write-only socket option to set role for DTLS connection. This option is irrelevant for TLS connections, as for them role is selected based on connect()/listen() usage. By default, DTLS will assume client role. This option accepts an integer with a TLS role, compatible with mbedTLS values:

- 0 - client
- 1 - server

TLS_ALPN_LIST

Socket option for setting the supported Application Layer Protocols. It accepts and returns a const char array of NULL terminated strings representing the supported application layer protocols listed during the TLS handshake.

TLS_DTLS_HANDSHAKE_TIMEOUT_MIN

Socket option to set DTLS handshake timeout. The timeout starts at min, and upon retransmission the timeout is doubled util max is reached. Min and max arguments are separate options. The time unit is ms.

TLS_DTLS_HANDSHAKE_TIMEOUT_MAX

API Reference

BSD Sockets

group bsd_sockets

BSD Sockets compatible API.

Defines

ZSOCK_POLLIN

zsock_poll: Poll for readability

ZSOCK_POLLPRI

zsock_poll: Compatibility value, ignored

ZSOCK_POLLOUT

zsock_poll: Poll for writability

ZSOCK_POLLERR
: zsock_poll: Poll results in error condition (output value only)

ZSOCK_POLLHUP
: zsock_poll: Poll detected closed connection (output value only)

ZSOCK_POLLNVAL
: zsock_poll: Invalid socket (output value only)

ZSOCK_MSG_PEEK
: zsock_recv: Read data without removing it from socket input queue

ZSOCK_MSG_TRUNC
: zsock_recv: return the real length of the datagram, even when it was longer than the passed buffer

ZSOCK_MSG_DONTWAIT
: zsock_recv/zsock_send: Override operation to non-blocking

ZSOCK_MSG_WAITALL
: zsock_recv: block until the full amount of data can be returned

ZSOCK_SHUT_RD
: zsock_shutdown: Shut down for reading

ZSOCK_SHUT_WR
: zsock_shutdown: Shut down for writing

ZSOCK_SHUT_RDWR
: zsock_shutdown: Shut down for both reading and writing

SOL_TLS
: Protocol level for TLS. Here, the same socket protocol level for TLS as in Linux was used.

TLS_PEER_VERIFY_NONE
: Peer verification disabled.

TLS_PEER_VERIFY_OPTIONAL
: Peer verification optional.

TLS_PEER_VERIFY_REQUIRED
: Peer verification required.

TLS_DTLS_ROLE_CLIENT
: Client role in a DTLS session.

TLS_DTLS_ROLE_SERVER
: Server role in a DTLS session.

`AI_PASSIVE`
: Address for bind() (vs for connect())

`AI_CANONNAME`
: Fill in ai_canonname

`AI_NUMERICHOST`
: Assume host address is in numeric notation, don't DNS lookup

`AI_V4MAPPED`
: May return IPv4 mapped address for IPv6

`AI_ALL`
: May return both native IPv6 and mapped IPv4 address for IPv6

`AI_ADDRCONFIG`
: IPv4/IPv6 support depends on local system config

`AI_NUMERICSERV`
: Assume service (port) is numeric

`NI_NUMERICHOST`
: *zsock_getnameinfo()*: Resolve to numeric address.

`NI_NUMERICSERV`
: *zsock_getnameinfo()*: Resolve to numeric port number.

`NI_NOFQDN`
: *zsock_getnameinfo()*: Return only hostname instead of FQDN

`NI_NAMEREQD`
: *zsock_getnameinfo()*: Dummy option for compatibility

`NI_DGRAM`
: *zsock_getnameinfo()*: Dummy option for compatibility

`NI_MAXHOST`
: *zsock_getnameinfo()*: Max supported hostname length

`IFNAMSIZ`

`SOL_SOCKET`
: sockopt: Socket-level option

`SO_REUSEADDR`
: sockopt: Enable server address reuse (ignored, for compatibility)

`SO_TYPE`
: sockopt: Type of the socket

SO_ERROR
: sockopt: Async error (ignored, for compatibility)

SO_RCVTIMEO
: sockopt: Receive timeout Applies to receive functions like recv(), but not to connect()

SO_SNDTIMEO
: sockopt: Send timeout

SO_BINDTODEVICE
: sockopt: Bind a socket to an interface

SO_TIMESTAMPING
: sockopt: Timestamp TX packets

SO_PROTOCOL
: sockopt: Protocol used with the socket

TCP_NODELAY
: sockopt: Disable TCP buffering (ignored, for compatibility)

IPV6_V6ONLY
: sockopt: Don't support IPv4 access (ignored, for compatibility)

SO_PRIORITY
: sockopt: Socket priority

SO_TXTIME
: sockopt: Socket TX time (when the data should be sent)

SCM_TXTIME

SO_SOCKS5
: sockopt: Enable SOCKS5 for Socket

ZSOCK_FD_SETSIZE
: Number of file descriptors which can be added to *zsock_fd_set*

zsock_timeval

Typedefs

typedef struct *zsock_fd_set* zsock_fd_set

Functions

void *zsock_get_context_object(int sock)

Obtain a file descriptor's associated net context.

With CONFIG_USERSPACE enabled, the kernel's object permission system must apply to socket file descriptors. When a socket is opened, by default only the caller has permission, access by other threads will fail unless they have been specifically granted permission.

This is achieved by tagging data structure definitions that implement the underlying object associated with a network socket file descriptor with '__net_socket`. All pointers to instances of these will be known to the kernel as kernel objects with type K_OBJ_NET_SOCKET.

This API is intended for threads that need to grant access to the object associated with a particular file descriptor to another thread. The returned pointer represents the underlying K_OBJ_NET_SOCKET and may be passed to APIs like *k_object_access_grant()*.

In a system like Linux which has the notion of threads running in processes in a shared virtual address space, this sort of management is unnecessary as the scope of file descriptors is implemented at the process level.

However in Zephyr the file descriptor scope is global, and MPU-based systems are not able to implement a process-like model due to the lack of memory virtualization hardware. They use discrete object permissions and memory domains instead to define thread access scope.

User threads will have no direct access to the returned object and will fault if they try to access its memory; the pointer can only be used to make permission assignment calls, which follow exactly the rules for other kernel objects like device drivers and IPC.

Parameters

- `sock` – file descriptor

Returns pointer to associated network socket object, or NULL if the file descriptor wasn't valid or the caller had no access permission

int zsock_socket(int family, int type, int proto)

Create a network socket.

See POSIX.1-2017 article for normative description. This function is also exposed as `socket()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

If CONFIG_USERSPACE is enabled, the caller will be granted access to the context object associated with the returned file descriptor.

See also:

zsock_get_context_object()

int zsock_socketpair(int family, int type, int proto, int *sv)

Create an unnamed pair of connected sockets.

See POSIX.1-2017 article for normative description. This function is also exposed as `socketpair()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int zsock_close(int sock)

Close a network socket.

Close a network socket. This function is also exposed as `close()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined (in which case it may conflict with generic POSIX `close()` function).

int zsock_shutdown(int sock, int how)

Shutdown socket send/receive operations.

See POSIX.1-2017 article for normative description, but currently this function has no effect in Zephyr and provided solely for compatibility with existing code. This function is also exposed as `shutdown()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int `zsock_bind`(int sock, const struct *sockaddr* *addr, *socklen_t* addrlen)

Bind a socket to a local network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `bind()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int `zsock_connect`(int sock, const struct *sockaddr* *addr, *socklen_t* addrlen)

Connect a socket to a peer network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `connect()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int `zsock_listen`(int sock, int backlog)

Set up a STREAM socket to accept peer connections.

See POSIX.1-2017 article for normative description. This function is also exposed as `listen()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int `zsock_accept`(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)

Accept a connection on listening socket.

See POSIX.1-2017 article for normative description. This function is also exposed as `accept()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

ssize_t `zsock_sendto`(int sock, const void *buf, size_t len, int flags, const struct *sockaddr* *dest_addr, *socklen_t* addrlen)

Send data to an arbitrary network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `sendto()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

static inline ssize_t `zsock_send`(int sock, const void *buf, size_t len, int flags)

Send data to a connected peer.

See POSIX.1-2017 article for normative description. This function is also exposed as `send()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

ssize_t `zsock_sendmsg`(int sock, const struct *msghdr* *msg, int flags)

Send data to an arbitrary network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `sendmsg()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

ssize_t `zsock_recvfrom`(int sock, void *buf, size_t max_len, int flags, struct *sockaddr* *src_addr, *socklen_t* *addrlen)

Receive data from an arbitrary network address.

See POSIX.1-2017 article for normative description. This function is also exposed as `recvfrom()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

static inline ssize_t `zsock_recv`(int sock, void *buf, size_t max_len, int flags)

Receive data from a connected peer.

See POSIX.1-2017 article for normative description. This function is also exposed as `recv()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int `zsock_fcntl`(int sock, int cmd, int flags)

Control blocking/non-blocking mode of a socket.

This functions allow to (only) configure a socket for blocking or non-blocking operation (O_NONBLOCK). This function is also exposed as `fcntl()` if

CONFIG_NET_SOCKETS_POSIX_NAMES is defined (in which case it may conflict with generic POSIX fcntl() function).

int zsock_poll(struct *zsock_pollfd* *fds, int nfds, int timeout)

Efficiently poll multiple sockets for events.

See POSIX.1-2017 article for normative description. (In Zephyr this function works only with sockets, not arbitrary file descriptors.) This function is also exposed as poll() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined (in which case it may conflict with generic POSIX poll() function).

int zsock_getsockopt(int sock, int level, int optname, void *optval, *socklen_t* *optlen)

Get various socket options.

See POSIX.1-2017 article for normative description. In Zephyr this function supports a subset of socket options described by POSIX, but also some additional options available in Linux (some options are dummy and provided to ease porting of existing code). This function is also exposed as getsockopt() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

int zsock_setsockopt(int sock, int level, int optname, const void *optval, *socklen_t* optlen)

Set various socket options.

See POSIX.1-2017 article for normative description. In Zephyr this function supports a subset of socket options described by POSIX, but also some additional options available in Linux (some options are dummy and provided to ease porting of existing code). This function is also exposed as setsockopt() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

int zsock_getsockname(int sock, struct *sockaddr* *addr, *socklen_t* *addrlen)

Get socket name.

See POSIX.1-2017 article for normative description. This function is also exposed as getsockname() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

int zsock_gethostname(char *buf, size_t len)

Get local host name.

See POSIX.1-2017 article for normative description. This function is also exposed as gethostname() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

static inline char *zsock_inet_ntop(*sa_family_t* family, const void *src, char *dst, size_t size)

Convert network address from internal to numeric ASCII form.

See POSIX.1-2017 article for normative description. This function is also exposed as inet_ntop() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

int zsock_inet_pton(*sa_family_t* family, const char *src, void *dst)

Convert network address from numeric ASCII form to internal representation.

See POSIX.1-2017 article for normative description. This function is also exposed as inet_pton() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

int zsock_getaddrinfo(const char *host, const char *service, const struct *zsock_addrinfo* *hints, struct *zsock_addrinfo* **res)

Resolve a domain name to one or more network addresses.

See POSIX.1-2017 article for normative description. This function is also exposed as getaddrinfo() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

void zsock_freeaddrinfo(struct *zsock_addrinfo* *ai)

Free results returned by *zsock_getaddrinfo()*

See POSIX.1-2017 article for normative description. This function is also exposed as freeaddrinfo() if CONFIG_NET_SOCKETS_POSIX_NAMES is defined.

const char *zsock_gai_strerror(int errcode)

Convert *zsock_getaddrinfo()* error code to textual message.

See POSIX.1-2017 article for normative description. This function is also exposed as `gai_strerror()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int zsock_getnameinfo(const struct *sockaddr* *addr, *socklen_t* addrlen, char *host, *socklen_t* hostlen, char *serv, *socklen_t* servlen, int flags)

Resolve a network address to a domain name or ASCII address.

See POSIX.1-2017 article for normative description. This function is also exposed as `getnameinfo()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int zsock_select(int nfds, *zsock_fd_set* *readfds, *zsock_fd_set* *writefds, *zsock_fd_set* *exceptfds, struct *zsock_timeval* *timeout)

Legacy function to poll multiple sockets for events.

See POSIX.1-2017 article for normative description. This function is provided to ease porting of existing code and not recommended for usage due to its inefficiency, use *zsock_poll()* instead. In Zephyr this function works only with sockets, not arbitrary file descriptors. This function is also exposed as `select()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined (in which case it may conflict with generic POSIX `select()` function).

void ZSOCK_FD_ZERO(*zsock_fd_set* *set)

Initialize (clear) fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_ZERO()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

int ZSOCK_FD_ISSET(int fd, *zsock_fd_set* *set)

Check whether socket is a member of fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_ISSET()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

void ZSOCK_FD_CLR(int fd, *zsock_fd_set* *set)

Remove socket from fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_CLR()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

void ZSOCK_FD_SET(int fd, *zsock_fd_set* *set)

Add socket to fd_set.

See POSIX.1-2017 article for normative description. This function is also exposed as `FD_SET()` if `CONFIG_NET_SOCKETS_POSIX_NAMES` is defined.

struct zsock_pollfd

#include <socket.h>

struct zsock_addrinfo

#include <socket.h>

struct ifreq

#include <socket.h> Interface description structure

struct zsock_fd_set

#include <socket_select.h>

TLS Credentials

group `tls_credentials`

TLS credentials management.

Typedefs

typedef int `sec_tag_t`

Secure tag, a reference to TLS credential

Secure tag can be used to reference credential after it was registered in the system.

Note: Some TLS credentials come in pairs:

- TLS_CREDENTIAL_SERVER_CERTIFICATE with TLS_CREDENTIAL_PRIVATE_KEY,
- TLS_CREDENTIAL_PSK with TLS_CREDENTIAL_PSK_ID. Such pairs of credentials must be assigned the same secure tag to be correctly handled in the system.

Enums

enum `tls_credential_type`

TLS credential types

Values:

enumerator `TLS_CREDENTIAL_NONE`

Unspecified credential.

enumerator `TLS_CREDENTIAL_CA_CERTIFICATE`

A trusted CA certificate. Use this to authenticate remote servers. Used with certificate-based ciphersuites.

enumerator `TLS_CREDENTIAL_SERVER_CERTIFICATE`

A public server certificate. Use this to register your own server certificate. Should be registered together with a corresponding private key. Used with certificate-based ciphersuites.

enumerator `TLS_CREDENTIAL_PRIVATE_KEY`

Private key. Should be registered together with a corresponding public certificate. Used with certificate-based ciphersuites.

enumerator `TLS_CREDENTIAL_PSK`

Pre-shared key. Should be registered together with a corresponding PSK identity. Used with PSK-based ciphersuites.

enumerator `TLS_CREDENTIAL_PSK_ID`

Pre-shared key identity. Should be registered together with a corresponding PSK. Used with PSK-based ciphersuites.

Functions

int `tls_credential_add`(*sec_tag_t* tag, enum *tls_credential_type* type, const void *cred, size_t credlen)

Add a TLS credential.

This function adds a TLS credential, that can be used by TLS/DTLS for authentication.

Parameters

- `tag` – A security tag that credential will be referenced with.
- `type` – A TLS/DTLS credential type.
- `cred` – A TLS/DTLS credential.
- `credlen` – A TLS/DTLS credential length.

Return values

- `0` – TLS credential successfully added.
- `-EACCES` – Access to the TLS credential subsystem was denied.
- `-ENOMEM` – Not enough memory to add new TLS credential.
- `-EEXIST` – TLS credential of specific tag and type already exists.

int `tls_credential_get`(*sec_tag_t* tag, enum *tls_credential_type* type, void *cred, size_t *credlen)

Get a TLS credential.

This function gets an already registered TLS credential, referenced by `tag` secure tag of `type`.

Parameters

- `tag` – A security tag of requested credential.
- `type` – A TLS/DTLS credential type of requested credential.
- `cred` – A buffer for TLS/DTLS credential.
- `credlen` – A buffer size on input. TLS/DTLS credential length on output.

Return values

- `0` – TLS credential successfully obtained.
- `-EACCES` – Access to the TLS credential subsystem was denied.
- `-ENOENT` – Requested TLS credential was not found.
- `-EFBIG` – Requested TLS credential does not fit in the buffer provided.

int `tls_credential_delete`(*sec_tag_t* tag, enum *tls_credential_type* type)

Delete a TLS credential.

This function removes a TLS credential, referenced by `tag` secure tag of `type`.

Parameters

- `tag` – A security tag corresponding to removed credential.
- `type` – A TLS/DTLS credential type of removed credential.

Return values

- `0` – TLS credential successfully deleted.
- `-EACCES` – Access to the TLS credential subsystem was denied.
- `-ENOENT` – Requested TLS credential was not found.

IPv4/IPv6 Primitives and Helpers

Overview Miscellaneous defines and helper functions for IP addresses and IP protocols.

API Reference

group `ip_4_6`

IPv4/IPv6 primitives and helpers.

Defines

`PF_UNSPEC`

Unspecified protocol family.

`PF_INET`

IP protocol family version 4.

`PF_INET6`

IP protocol family version 6.

`PF_PACKET`

Packet family.

`PF_CAN`

Controller Area Network.

`PF_NET_MGMT`

Network management info.

`PF_LOCAL`

Inter-process communication

`PF_UNIX`

Inter-process communication

`AF_UNSPEC`

Unspecified address family.

`AF_INET`

IP protocol family version 4.

`AF_INET6`

IP protocol family version 6.

AF_PACKET
: Packet family.

AF_CAN
: Controller Area Network.

AF_NET_MGMT
: Network management info.

AF_LOCAL
: Inter-process communication

AF_UNIX
: Inter-process communication

ntohs(x)
: Convert 16-bit value from network to host byte order.

  Parameters

  - x – The network byte order value to convert.

  Returns Host byte order value.

ntohl(x)
: Convert 32-bit value from network to host byte order.

  Parameters

  - x – The network byte order value to convert.

  Returns Host byte order value.

ntohll(x)
: Convert 64-bit value from network to host byte order.

  Parameters

  - x – The network byte order value to convert.

  Returns Host byte order value.

htons(x)
: Convert 16-bit value from host to network byte order.

  Parameters

  - x – The host byte order value to convert.

  Returns Network byte order value.

htonl(x)
: Convert 32-bit value from host to network byte order.

  Parameters

  - x – The host byte order value to convert.

  Returns Network byte order value.

htonll(x)
: Convert 64-bit value from host to network byte order.

  Parameters

- `x` – The host byte order value to convert.

Returns Network byte order value.

`ALIGN_H(x)`

`ALIGN_D(x)`

`CMSG_FIRSTHDR(msghdr)`

`CMSG_NXTHDR(msghdr, cmsg)`

`CMSG_DATA(cmsg)`

`CMSG_SPACE(length)`

`CMSG_LEN(length)`

`INET_ADDRSTRLEN`

Max length of the IPv4 address as a string. Defined by POSIX.

`INET6_ADDRSTRLEN`

Max length of the IPv6 address as a string. Takes into account possible mapped IPv4 addresses.

`NET_MAX_PRIORITIES`

`net_ipaddr_copy(dest, src)`

Copy an IPv4 or IPv6 address.

Parameters

- `dest` – Destination IP address.
- `src` – Source IP address.

Returns Destination address.

Typedefs

typedef unsigned short int `sa_family_t`

Socket address family type

typedef size_t `socklen_t`

Length of a socket address

Enums

enum `net_ip_protocol`

Protocol numbers from IANA/BSD

Values:

enumerator `IPPROTO_IP` = 0

IP protocol (pseudo-val for setsockopt()

enumerator IPPROTO_ICMP = 1
ICMP protocol

enumerator IPPROTO_IGMP = 2
IGMP protocol

enumerator IPPROTO_IPIP = 4
IPIP tunnels

enumerator IPPROTO_TCP = 6
TCP protocol

enumerator IPPROTO_UDP = 17
UDP protocol

enumerator IPPROTO_IPV6 = 41
IPv6 protocol

enumerator IPPROTO_ICMPV6 = 58
ICMPv6 protocol

enumerator IPPROTO_RAW = 255
RAW IP packets

enum net_ip_protocol_secure
Protocol numbers for TLS protocols
Values:

enumerator IPPROTO_TLS_1_0 = 256
TLS 1.0 protocol

enumerator IPPROTO_TLS_1_1 = 257
TLS 1.1 protocol

enumerator IPPROTO_TLS_1_2 = 258
TLS 1.2 protocol

enumerator IPPROTO_DTLS_1_0 = 272
DTLS 1.0 protocol

enumerator IPPROTO_DTLS_1_2 = 273
DTLS 1.2 protocol

enum net_sock_type
Socket type
Values:

enumerator SOCK_STREAM = 1
Stream socket type

enumerator SOCK_DGRAM
Datagram socket type

enumerator SOCK_RAW
RAW socket type

enum net_ip_mtu
Values:

enumerator NET_IPV6_MTU = 1280
IPv6 MTU length. We must be able to receive this size IPv6 packet without fragmentation.

enumerator NET_IPV4_MTU = 576
IPv4 MTU length. We must be able to receive this size IPv4 packet without fragmentation.

enum net_priority
Network packet priority settings described in IEEE 802.1Q Annex I.1
Values:

enumerator NET_PRIORITY_BK = 1
Background (lowest)

enumerator NET_PRIORITY_BE = 0
Best effort (default)

enumerator NET_PRIORITY_EE = 2
Excellent effort

enumerator NET_PRIORITY_CA = 3
Critical applications (highest)

enumerator NET_PRIORITY_VI = 4
Video, < 100 ms latency and jitter

enumerator NET_PRIORITY_VO = 5
Voice, < 10 ms latency and jitter

enumerator NET_PRIORITY_IC = 6
Internetwork control

enumerator NET_PRIORITY_NC = 7
Network control

enum net_addr_state
What is the current state of the network address
Values:

enumerator NET_ADDR_ANY_STATE = -1

Default (invalid) address type

enumerator NET_ADDR_TENTATIVE = 0

Tentative address

enumerator NET_ADDR_PREFERRED

Preferred address

enumerator NET_ADDR_DEPRECATED

Deprecated address

enum net_addr_type

How the network address is assigned to network interface

Values:

enumerator NET_ADDR_ANY = 0

Default value. This is not a valid value.

enumerator NET_ADDR_AUTOCONF

Auto configured address

enumerator NET_ADDR_DHCP

Address is from DHCP

enumerator NET_ADDR_MANUAL

Manually set address

enumerator NET_ADDR_OVERRIDABLE

Manually set address which is overridable by DHCP

Functions

static inline bool net_ipv6_is_addr_loopback(struct *in6_addr* *addr)

Check if the IPv6 address is a loopback address (::1).

Parameters

- addr – IPv6 address

Returns True if address is a loopback address, False otherwise.

static inline bool net_ipv6_is_addr_mcast(const struct *in6_addr* *addr)

Check if the IPv6 address is a multicast address.

Parameters

- addr – IPv6 address

Returns True if address is multicast address, False otherwise.

struct *net_if_addr* *net_if_ipv6_addr_lookup(const struct *in6_addr* *addr, struct *net_if* **iface)

static inline bool net_ipv6_is_my_addr(struct *in6_addr* *addr)

Check if IPv6 address is found in one of the network interfaces.

Parameters

- addr – IPv6 address

Returns True if address was found, False otherwise.

struct *net_if_mcast_addr* *net_if_ipv6_maddr_lookup(const struct *in6_addr* *addr, struct *net_if* **iface)

static inline bool net_ipv6_is_my_maddr(struct *in6_addr* *maddr)

Check if IPv6 multicast address is found in one of the network interfaces.

Parameters

- maddr – Multicast IPv6 address

Returns True if address was found, False otherwise.

static inline bool net_ipv6_is_prefix(const uint8_t *addr1, const uint8_t *addr2, uint8_t length)

Check if two IPv6 addresses are same when compared after prefix mask.

Parameters

- addr1 – First IPv6 address.
- addr2 – Second IPv6 address.
- length – Prefix length (max length is 128).

Returns True if IPv6 prefixes are the same, False otherwise.

static inline bool net_ipv4_is_addr_loopback(struct *in_addr* *addr)

Check if the IPv4 address is a loopback address (127.0.0.0/8).

Parameters

- addr – IPv4 address

Returns True if address is a loopback address, False otherwise.

static inline bool net_ipv4_is_addr_unspecified(const struct *in_addr* *addr)

Check if the IPv4 address is unspecified (all bits zero)

Parameters

- addr – IPv4 address.

Returns True if the address is unspecified, false otherwise.

static inline bool net_ipv4_is_addr_mcast(const struct *in_addr* *addr)

Check if the IPv4 address is a multicast address.

Parameters

- addr – IPv4 address

Returns True if address is multicast address, False otherwise.

static inline bool net_ipv4_is_ll_addr(const struct *in_addr* *addr)

Check if the given IPv4 address is a link local address.

Parameters

- addr – A valid pointer on an IPv4 address

Returns True if it is, false otherwise.

static inline bool net_ipv4_addr_cmp(const struct *in_addr* *addr1, const struct *in_addr* *addr2)

Compare two IPv4 addresses.

Parameters

- addr1 – Pointer to IPv4 address.
- addr2 – Pointer to IPv4 address.

Returns True if the addresses are the same, false otherwise.

static inline bool net_ipv6_addr_cmp(const struct *in6_addr* *addr1, const struct *in6_addr* *addr2)

Compare two IPv6 addresses.

Parameters

- addr1 – Pointer to IPv6 address.
- addr2 – Pointer to IPv6 address.

Returns True if the addresses are the same, false otherwise.

static inline bool net_ipv6_is_ll_addr(const struct *in6_addr* *addr)

Check if the given IPv6 address is a link local address.

Parameters

- addr – A valid pointer on an IPv6 address

Returns True if it is, false otherwise.

static inline bool net_ipv6_is_ula_addr(const struct *in6_addr* *addr)

Check if the given IPv6 address is a unique local address.

Parameters

- addr – A valid pointer on an IPv6 address

Returns True if it is, false otherwise.

const struct *in6_addr* *net_ipv6_unspecified_address(void)

Return pointer to any (all bits zeros) IPv6 address.

Returns Any IPv6 address.

const struct *in_addr* *net_ipv4_unspecified_address(void)

Return pointer to any (all bits zeros) IPv4 address.

Returns Any IPv4 address.

const struct *in_addr* *net_ipv4_broadcast_address(void)

Return pointer to broadcast (all bits ones) IPv4 address.

Returns Broadcast IPv4 address.

bool net_if_ipv4_addr_mask_cmp(struct *net_if* *iface, const struct *in_addr* *addr)

static inline bool net_ipv4_addr_mask_cmp(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given address belongs to same subnet that has been configured for the interface.

Parameters

- iface – A valid pointer on an interface
- addr – IPv4 address

Returns True if address is in same subnet, false otherwise.

bool net_if_ipv4_is_addr_bcast(struct *net_if* *iface, const struct *in_addr* *addr)

static inline bool net_ipv4_is_addr_bcast(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given IPv4 address is a broadcast address.

Parameters

- iface – Interface to use. Must be a valid pointer to an interface.
- addr – IPv4 address

Returns True if address is a broadcast address, false otherwise.

struct *net_if_addr* *net_if_ipv4_addr_lookup(const struct *in_addr* *addr, struct *net_if* **iface)

static inline bool net_ipv4_is_my_addr(const struct *in_addr* *addr)

Check if the IPv4 address is assigned to any network interface in the system.

Parameters

- addr – A valid pointer on an IPv4 address

Returns True if IPv4 address is found in one of the network interfaces, False otherwise.

static inline bool net_ipv6_is_addr_unspecified(const struct *in6_addr* *addr)

Check if the IPv6 address is unspecified (all bits zero)

Parameters

- addr – IPv6 address.

Returns True if the address is unspecified, false otherwise.

static inline bool net_ipv6_is_addr_solicited_node(const struct *in6_addr* *addr)

Check if the IPv6 address is solicited node multicast address FF02:0:0:0:0:1:FFXX:XXXX defined in RFC 3513.

Parameters

- addr – IPv6 address.

Returns True if the address is solicited node address, false otherwise.

static inline bool net_ipv6_is_addr_mcast_scope(const struct *in6_addr* *addr, int scope)

Check if the IPv6 address is a given scope multicast address (FFyx::).

Parameters

- addr – IPv6 address
- scope – Scope to check

Returns True if the address is in given scope multicast address, false otherwise.

static inline bool net_ipv6_is_same_mcast_scope(const struct *in6_addr* *addr_1, const struct *in6_addr* *addr_2)

Check if the IPv6 addresses have the same multicast scope (FFyx::).

Parameters

- addr_1 – IPv6 address 1
- addr_2 – IPv6 address 2

Returns True if both addresses have same multicast scope, false otherwise.

static inline bool net_ipv6_is_addr_mcast_global(const struct *in6_addr* *addr)

Check if the IPv6 address is a global multicast address (FFxE::/16).

Parameters

- addr – IPv6 address.

Returns True if the address is global multicast address, false otherwise.

static inline bool `net_ipv6_is_addr_mcast_iface`(const struct *in6_addr* *addr)

Check if the IPv6 address is a interface scope multicast address (FFx1::).

Parameters

- `addr` – IPv6 address.

Returns True if the address is a interface scope multicast address, false otherwise.

static inline bool `net_ipv6_is_addr_mcast_link`(const struct *in6_addr* *addr)

Check if the IPv6 address is a link local scope multicast address (FFx2::).

Parameters

- `addr` – IPv6 address.

Returns True if the address is a link local scope multicast address, false otherwise.

static inline bool `net_ipv6_is_addr_mcast_mesh`(const struct *in6_addr* *addr)

Check if the IPv6 address is a mesh-local scope multicast address (FFx3::).

Parameters

- `addr` – IPv6 address.

Returns True if the address is a mesh-local scope multicast address, false otherwise.

static inline bool `net_ipv6_is_addr_mcast_site`(const struct *in6_addr* *addr)

Check if the IPv6 address is a site scope multicast address (FFx5::).

Parameters

- `addr` – IPv6 address.

Returns True if the address is a site scope multicast address, false otherwise.

static inline bool `net_ipv6_is_addr_mcast_org`(const struct *in6_addr* *addr)

Check if the IPv6 address is an organization scope multicast address (FFx8::).

Parameters

- `addr` – IPv6 address.

Returns True if the address is an organization scope multicast address, false otherwise.

static inline bool `net_ipv6_is_addr_mcast_group`(const struct *in6_addr* *addr, const struct *in6_addr* *group)

Check if the IPv6 address belongs to certain multicast group.

Parameters

- `addr` – IPv6 address.
- `group` – Group id IPv6 address, the values must be in network byte order

Returns True if the IPv6 multicast address belongs to given multicast group, false otherwise.

static inline bool `net_ipv6_is_addr_mcast_all_nodes_group`(const struct *in6_addr* *addr)

Check if the IPv6 address belongs to the all nodes multicast group.

Parameters

- `addr` – IPv6 address

Returns True if the IPv6 multicast address belongs to the all nodes multicast group, false otherwise

static inline bool net_ipv6_is_addr_mcast_iface_all_nodes(const struct *in6_addr* *addr)

Check if the IPv6 address is a interface scope all nodes multicast address (FF01::1).

Parameters

- addr – IPv6 address.

Returns True if the address is a interface scope all nodes multicast address, false otherwise.

static inline bool net_ipv6_is_addr_mcast_link_all_nodes(const struct *in6_addr* *addr)

Check if the IPv6 address is a link local scope all nodes multicast address (FF02::1).

Parameters

- addr – IPv6 address.

Returns True if the address is a link local scope all nodes multicast address, false otherwise.

static inline void net_ipv6_addr_create_solicited_node(const struct *in6_addr* *src, struct *in6_addr* *dst)

Create solicited node IPv6 multicast address FF02:0:0:0:0:1:FFXX:XXXX defined in RFC 3513.

Parameters

- src – IPv6 address.
- dst – IPv6 address.

static inline void net_ipv6_addr_create(struct *in6_addr* *addr, uint16_t addr0, uint16_t addr1, uint16_t addr2, uint16_t addr3, uint16_t addr4, uint16_t addr5, uint16_t addr6, uint16_t addr7)

Construct an IPv6 address from eight 16-bit words.

Parameters

- addr – IPv6 address
- addr0 – 16-bit word which is part of the address
- addr1 – 16-bit word which is part of the address
- addr2 – 16-bit word which is part of the address
- addr3 – 16-bit word which is part of the address
- addr4 – 16-bit word which is part of the address
- addr5 – 16-bit word which is part of the address
- addr6 – 16-bit word which is part of the address
- addr7 – 16-bit word which is part of the address

static inline void net_ipv6_addr_create_ll_allnodes_mcast(struct *in6_addr* *addr)

Create link local allnodes multicast IPv6 address.

Parameters

- addr – IPv6 address

static inline void net_ipv6_addr_create_ll_allrouters_mcast(struct *in6_addr* *addr)

Create link local allrouters multicast IPv6 address.

Parameters

- addr – IPv6 address

static inline void net_ipv6_addr_create_iid(struct *in6_addr* *addr, struct *net_linkaddr* *lladdr)

Create IPv6 address interface identifier.

Parameters

- addr – IPv6 address
- lladdr – Link local address

static inline bool net_ipv6_addr_based_on_ll(const struct *in6_addr* *addr, const struct *net_linkaddr* *lladdr)

Check if given address is based on link layer address.

Returns True if it is, False otherwise

static inline struct *sockaddr_in6* *net_sin6(const struct *sockaddr* *addr)

Get *sockaddr_in6* from sockaddr. This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns Pointer to IPv6 socket address

static inline struct *sockaddr_in* *net_sin(const struct *sockaddr* *addr)

Get *sockaddr_in* from sockaddr. This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns Pointer to IPv4 socket address

static inline struct *sockaddr_in6_ptr* *net_sin6_ptr(const struct sockaddr_ptr *addr)

Get *sockaddr_in6_ptr* from sockaddr_ptr. This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns Pointer to IPv6 socket address

static inline struct *sockaddr_in_ptr* *net_sin_ptr(const struct sockaddr_ptr *addr)

Get *sockaddr_in_ptr* from sockaddr_ptr. This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns Pointer to IPv4 socket address

static inline struct *sockaddr_ll_ptr* *net_sll_ptr(const struct sockaddr_ptr *addr)

Get *sockaddr_ll_ptr* from sockaddr_ptr. This is a helper so that the code calling this function can be made shorter.

Parameters

- addr – Socket address

Returns Pointer to linklayer socket address

static inline struct *sockaddr_can_ptr* *net_can_ptr(const struct sockaddr_ptr *addr)

Get *sockaddr_can_ptr* from sockaddr_ptr. This is a helper so that the code needing this functionality can be made shorter.

Parameters

- `addr` – Socket address

Returns Pointer to CAN socket address

int `net_addr_pton`(*sa_family_t* family, const char *src, void *dst)

Convert a string to IP address.

Note: This function doesn't do precise error checking, do not use for untrusted strings.

Parameters

- `family` – IP address family (AF_INET or AF_INET6)
- `src` – IP address in a null terminated string
- `dst` – Pointer to struct *in_addr* if family is AF_INET or pointer to struct *in6_addr* if family is AF_INET6

Returns 0 if ok, < 0 if error

char *`net_addr_ntop`(*sa_family_t* family, const void *src, char *dst, size_t size)

Convert IP address to string form.

Parameters

- `family` – IP address family (AF_INET or AF_INET6)
- `src` – Pointer to struct *in_addr* if family is AF_INET or pointer to struct *in6_addr* if family is AF_INET6
- `dst` – Buffer for IP address as a null terminated string
- `size` – Number of bytes available in the buffer

Returns dst pointer if ok, NULL if error

bool `net_ipaddr_parse`(const char *str, size_t str_len, struct *sockaddr* *addr)

Parse a string that contains either IPv4 or IPv6 address and optional port, and store the information in user supplied sockaddr struct.

Syntax of the IP address string: 192.0.2.1:80 192.0.2.42

[2001:db8::2] 2001:db::42 Note that the str_len parameter is used to restrict the amount of characters that are checked. If the string does not contain port number, then the port number in sockaddr is not modified.

Parameters

- `str` – String that contains the IP address.
- `str_len` – Length of the string to be parsed.
- `addr` – Pointer to user supplied struct sockaddr.

Returns True if parsing could be done, false otherwise.

static inline int32_t `net_tcp_seq_cmp`(uint32_t seq1, uint32_t seq2)

Compare TCP sequence numbers.

This function compares TCP sequence numbers, accounting for wraparound effects.

Parameters

- `seq1` – First sequence number
- `seq2` – Seconds sequence number

Returns < 0 if seq1 < seq2, 0 if seq1 == seq2, > 0 if seq > seq2

static inline bool net_tcp_seq_greater(uint32_t seq1, uint32_t seq2)

Check that one TCP sequence number is greater.

This is convenience function on top of *net_tcp_seq_cmp()*.

Parameters

- seq1 – First sequence number
- seq2 – Seconds sequence number

Returns True if seq > seq2

int net_bytes_from_str(uint8_t *buf, int buf_len, const char *src)

Convert a string of hex values to array of bytes.

The syntax of the string is “ab:02:98:fa:42:01”

Parameters

- buf – Pointer to memory where the bytes are written.
- buf_len – Length of the memory area.
- src – String of bytes.

Returns 0 if ok, <0 if error

int net_tx_priority2tc(enum *net_priority* prio)

Convert Tx network packet priority to traffic class so we can place the packet into correct Tx queue.

Parameters

- prio – Network priority

Returns Tx traffic class that handles that priority network traffic.

int net_rx_priority2tc(enum *net_priority* prio)

Convert Rx network packet priority to traffic class so we can place the packet into correct Rx queue.

Parameters

- prio – Network priority

Returns Rx traffic class that handles that priority network traffic.

static inline enum *net_priority* net_vlan2priority(uint8_t priority)

Convert network packet VLAN priority to network packet priority so we can place the packet into correct queue.

Parameters

- priority – VLAN priority

Returns Network priority

static inline uint8_t net_priority2vlan(enum *net_priority* priority)

Convert network packet priority to network packet VLAN priority.

Parameters

- priority – Packet priority

Returns VLAN priority (PCP)

const char *net_family2str(*sa_family_t* family)

Return network address family value as a string. This is only usable for debugging.

Parameters

- `family` – Network address family code

Returns Network address family as a string, or NULL if family is unknown.

struct `in6_addr`

#include <net_ip.h> IPv6 address struct

struct `in_addr`

#include <net_ip.h> IPv4 address struct

struct `sockaddr_in6`

#include <net_ip.h> Socket address struct for IPv6.

struct `sockaddr_in6_ptr`

#include <net_ip.h>

struct `sockaddr_in`

#include <net_ip.h> Socket address struct for IPv4.

struct `sockaddr_in_ptr`

#include <net_ip.h>

struct `sockaddr_ll`

#include <net_ip.h> Socket address struct for packet socket.

struct `sockaddr_ll_ptr`

#include <net_ip.h>

struct `sockaddr_can_ptr`

#include <net_ip.h>

struct `iovec`

#include <net_ip.h>

struct `msghdr`

#include <net_ip.h>

struct `cmsghdr`

#include <net_ip.h>

struct `sockaddr`

#include <net_ip.h> Generic sockaddr struct. Must be cast to proper type.

struct `net_tuple`

#include <net_ip.h> IPv6/IPv4 network connection tuple

Public Members

struct net_addr *remote_addr
: IPv6/IPv4 remote address

struct net_addr *local_addr
: IPv6/IPv4 local address

uint16_t remote_port
: UDP/TCP remote port

uint16_t local_port
: UDP/TCP local port

enum *net_ip_protocol* ip_proto
: IP protocol

DNS Resolve

Overview The DNS resolver implements a basic DNS resolver according to IETF RFC1035 on Domain Implementation and Specification. Supported DNS answers are IPv4/IPv6 addresses and CNAME.

If a CNAME is received, the DNS resolver will create another DNS query. The number of additional queries is controlled by the CONFIG_DNS_RESOLVER_ADDITIONAL_QUERIES Kconfig variable.

The multicast DNS (mDNS) client resolver support can be enabled by setting CONFIG_MDNS_RESOLVER Kconfig option. See IETF RFC6762 for more details about mDNS.

The link-local multicast name resolution (LLMNR) client resolver support can be enabled by setting the CONFIG_LLMNR_RESOLVER Kconfig variable. See IETF RFC4795 for more details about LLMNR.

For more information about DNS configuration variables, see: subsys/net/lib/dns/Kconfig. The DNS resolver API can be found at include/net/dns_resolve.h.

Sample usage See DNS resolve sample application for details.

API Reference

group dns_resolve
: DNS resolving library.

Defines

DNS_MAX_NAME_SIZE
: Max size of the resolved name.

Typedefs

typedef void (*dns_resolve_cb_t)(enum *dns_resolve_status* status, struct *dns_addrinfo* *info, void *user_data)

DNS resolve callback.

The DNS resolve callback is called after a successful DNS resolving. The resolver can call this callback multiple times, one for each resolved address.

Param status The status of the query: DNS_EAI_INPROGRESS returned for each resolved address DNS_EAI_ALLDONE mark end of the resolving, info is set to NULL in this case DNS_EAI_CANCELED if the query was canceled manually or timeout happened DNS_EAI_FAIL if the name cannot be resolved by the server DNS_EAI_NODATA if there is no such name other values means that an error happened.

Param info Query results are stored here.

Param user_data The user data given in *dns_resolve_name()* call.

Enums

enum dns_query_type

DNS query type enum

Values:

enumerator DNS_QUERY_TYPE_A = 1

IPv4 query

enumerator DNS_QUERY_TYPE_AAAA = 28

IPv6 query

enum dns_resolve_status

Status values for the callback.

Values:

enumerator DNS_EAI_BADFLAGS = -1

Invalid value for ai_flags field

enumerator DNS_EAI_NONAME = -2

NAME or SERVICE is unknown

enumerator DNS_EAI_AGAIN = -3

Temporary failure in name resolution

enumerator DNS_EAI_FAIL = -4

Non-recoverable failure in name res

enumerator DNS_EAI_NODATA = -5

No address associated with NAME

enumerator DNS_EAI_FAMILY = -6
ai_family not supported

enumerator DNS_EAI_SOCKTYPE = -7
ai_socktype not supported

enumerator DNS_EAI_SERVICE = -8
SRV not supported for ai_socktype

enumerator DNS_EAI_ADDRFAMILY = -9
Address family for NAME not supported

enumerator DNS_EAI_MEMORY = -10
Memory allocation failure

enumerator DNS_EAI_SYSTEM = -11
System error returned in errno

enumerator DNS_EAI_OVERFLOW = -12
Argument buffer overflow

enumerator DNS_EAI_INPROGRESS = -100
Processing request in progress

enumerator DNS_EAI_CANCELED = -101
Request canceled

enumerator DNS_EAI_NOTCANCELED = -102
Request not canceled

enumerator DNS_EAI_ALLDONE = -103
All requests done

enumerator DNS_EAI_IDN_ENCODE = -105
IDN encoding failed

enum dns_resolve_context_state
Values:

enumerator DNS_RESOLVE_CONTEXT_ACTIVE

enumerator DNS_RESOLVE_CONTEXT_DEACTIVATING

enumerator DNS_RESOLVE_CONTEXT_INACTIVE

Functions

int dns_resolve_init(struct *dns_resolve_context* *ctx, const char *dns_servers_str[], const struct *sockaddr* *dns_servers_sa[])

Init DNS resolving context.

This function sets the DNS server address and initializes the DNS context that is used by the actual resolver. DNS server addresses can be specified either in textual form, or as struct sockaddr (or both). Note that the recommended way to resolve DNS names is to use the *dns_get_addr_info()* API. In that case user does not need to call *dns_resolve_init()* as the DNS servers are already setup by the system.

Parameters

- ctx – DNS context. If the context variable is allocated from the stack, then the variable needs to be valid for the whole duration of the resolving. Caller does not need to fill the variable beforehand or edit the context afterwards.
- dns_servers_str – DNS server addresses using textual strings. The array is NULL terminated. The port number can be given in the string. Syntax for the server addresses with or without port numbers: IPv4 : 10.0.9.1 IPv4 + port : 10.0.9.1:5353 IPv6 : 2001:db8::22:42 IPv6 + port : [2001:db8::22:42]:5353
- dns_servers_sa – DNS server addresses as struct sockaddr. The array is NULL terminated. Port numbers are optional in struct sockaddr, the default will be used if set to 0.

Returns 0 if ok, <0 if error.

int dns_resolve_close(struct *dns_resolve_context* *ctx)

Close DNS resolving context.

This releases DNS resolving context and marks the context unusable. Caller must call the *dns_resolve_init()* again to make context usable.

Parameters

- ctx – DNS context

Returns 0 if ok, <0 if error.

int dns_resolve_reconfigure(struct *dns_resolve_context* *ctx, const char *servers_str[], const struct *sockaddr* *servers_sa[])

Reconfigure DNS resolving context.

Reconfigures DNS context with new server list.

Parameters

- ctx – DNS context
- servers_str – DNS server addresses using textual strings. The array is NULL terminated. The port number can be given in the string. Syntax for the server addresses with or without port numbers: IPv4 : 10.0.9.1 IPv4 + port : 10.0.9.1:5353 IPv6 : 2001:db8::22:42 IPv6 + port : [2001:db8::22:42]:5353
- servers_sa – DNS server addresses as struct sockaddr. The array is NULL terminated. Port numbers are optional in struct sockaddr, the default will be used if set to 0.

Returns 0 if ok, <0 if error.

int dns_resolve_cancel(struct *dns_resolve_context* *ctx, uint16_t dns_id)

Cancel a pending DNS query.

This releases DNS resources used by a pending query.

Parameters

- ctx – DNS context

- `dns_id` – DNS id of the pending query

Returns 0 if ok, <0 if error.

int `dns_resolve_cancel_with_name`(struct *dns_resolve_context* *ctx, uint16_t dns_id, const char *query_name, enum *dns_query_type* query_type)

Cancel a pending DNS query using id, name and type.

This releases DNS resources used by a pending query.

Parameters

- `ctx` – DNS context
- `dns_id` – DNS id of the pending query
- `query_name` – Name of the resource we are trying to query (hostname)
- `query_type` – Type of the query (A or AAAA)

Returns 0 if ok, <0 if error.

int `dns_resolve_name`(struct *dns_resolve_context* *ctx, const char *query, enum *dns_query_type* type, uint16_t *dns_id, *dns_resolve_cb_t* cb, void *user_data, int32_t timeout)

Resolve DNS name.

This function can be used to resolve e.g., IPv4 or IPv6 address. Note that this is asynchronous call, the function will return immediately and system will call the callback after resolving has finished or timeout has occurred. We might send the query to multiple servers (if there are more than one server configured), but we only use the result of the first received response.

Parameters

- `ctx` – DNS context
- `query` – What the caller wants to resolve.
- `type` – What kind of data the caller wants to get.
- `dns_id` – DNS id is returned to the caller. This is needed if one wishes to cancel the query. This can be set to NULL if there is no need to cancel the query.
- `cb` – Callback to call after the resolving has finished or timeout has happened.
- `user_data` – The user data.
- `timeout` – The timeout value for the query. Possible values: SYS_FOREVER_MS: the query is tried forever, user needs to cancel it manually if it takes too long time to finish >0: start the query and let the system timeout it after specified ms

Returns 0 if resolving was started ok, < 0 otherwise

struct *dns_resolve_context* *`dns_resolve_get_default`(void)

Get default DNS context.

The system level DNS context uses DNS servers that are defined in project config file. If no DNS servers are defined by the user, then resolving DNS names using default DNS context will do nothing. The configuration options are described in subsys/net/lib/dns/Kconfig file.

Returns Default DNS context.

static inline int `dns_get_addr_info`(const char *query, enum *dns_query_type* type, uint16_t *dns_id, *dns_resolve_cb_t* cb, void *user_data, int32_t timeout)

Get IP address info from DNS.

This function can be used to resolve e.g., IPv4 or IPv6 address. Note that this is asynchronous call, the function will return immediately and system will call the callback after resolving has finished or timeout has occurred. We might send the query to multiple servers (if there are more than one server configured), but we only use the result of the first received response. This variant uses system wide DNS servers.

Parameters

- `query` – What the caller wants to resolve.
- `type` – What kind of data the caller wants to get.
- `dns_id` – DNS id is returned to the caller. This is needed if one wishes to cancel the query. This can be set to NULL if there is no need to cancel the query.
- `cb` – Callback to call after the resolving has finished or timeout has happened.
- `user_data` – The user data.
- `timeout` – The timeout value for the connection. Possible values: SYS_FOREVER_MS: the query is tried forever, user needs to cancel it manually if it takes too long time to finish >0: start the query and let the system timeout it after specified ms

Returns 0 if resolving was started ok, < 0 otherwise

static inline int `dns_cancel_addr_info(uint16_t dns_id)`

Cancel a pending DNS query.

This releases DNS resources used by a pending query.

Parameters

- `dns_id` – DNS id of the pending query

Returns 0 if ok, <0 if error.

struct `dns_addrinfo`

#include <dns_resolve.h> Address info struct is passed to callback that gets all the results.

struct `dns_resolve_context`

#include <dns_resolve.h> DNS resolve context structure.

Public Members

struct *sockaddr* `dns_server`

DNS server information

struct net_context *`net_ctx`

Connection to the DNS server

uint8_t `is_mdns`

Is this server mDNS one

uint8_t `is_llmnr`

Is this server LLMNR one

struct *k_mutex* `lock`

Prevent concurrent access

k_timeout_t `buf_timeout`

This timeout is also used when a buffer is required from the buffer pools.

enum *dns_resolve_context_state* `state`

Is this context in use

struct `dns_pending_query`

#include <dns_resolve.h> Result callbacks. We have multiple callbacks here so that it is possible to do multiple queries at the same time.

Contents of this structure can be inspected and changed only when the lock is held.

Public Members

struct *k_work_delayable* `timer`

Timeout timer

struct *dns_resolve_context* *`ctx`

Back pointer to ctx, needed in timeout handler

dns_resolve_cb_t `cb`

Result callback.

A null value indicates the slot is not in use.

void *`user_data`

User data

k_timeout_t `timeout`

TX timeout

const char *`query`

String containing the thing to resolve like www.example.com

This is set to a non-null value when the query is started, and is not used thereafter.

If the query completed at a point where the work item was still pending the pointer is cleared to indicate that the query is complete, but release of the query slot will be deferred until a request for a slot determines that the work item has been released.

enum *dns_query_type* `query_type`

Query type

uint16_t `id`

DNS id of this query

uint16_t `query_hash`

Hash of the DNS name + query type we are querying. This hash is calculated so we can match the response that we are receiving. This is needed mainly for mDNS which is setting the DNS id to 0, which means that the id alone cannot be used to find correct pending query.

Network Management

Overview The Network Management APIs allow applications, as well as network layer code itself, to call defined network routines at any level in the IP stack, or receive notifications on relevant network events. For example, by using these APIs, application code can request a scan be done on a Wi-Fi- or Bluetooth-based network interface, or request notification if a network interface IP address changes.

The Network Management API implementation is designed to save memory by eliminating code at build time for management routines that are not used. Distinct and statically defined APIs for network management procedures are not used. Instead, defined procedure handlers are registered by using a *NET_MGMT_REGISTER_REQUEST_HANDLER* macro. Procedure requests are done through a single *net_mgmt()* API that invokes the registered handler for the corresponding request.

The current implementation is experimental and may change and improve in future releases.

Requesting a defined procedure All network management requests are of the form `net_mgmt(mgmt_request, ...)`. The `mgmt_request` parameter is a bit mask that tells which stack layer is targeted, if a `net_if` object is implied, and the specific management procedure being requested. The available procedure requests depend on what has been implemented in the stack.

To avoid extra cost, all *net_mgmt()* calls are direct. Though this may change in a future release, it will not affect the users of this function.

Listening to network events You can receive notifications on network events by registering a callback function and specifying a set of events used to filter when your callback is invoked. The callback will have to be unique for a pair of layer and code, whereas on the command part it will be a mask of events.

Two functions are available, *net_mgmt_add_event_callback()* for registering the callback function, and *net_mgmt_del_event_callback()* for unregistering a callback. A helper function, *net_mgmt_init_event_callback()*, can be used to ease the initialization of the callback structure.

When an event occurs that matches a callback's event set, the associated callback function is invoked with the actual event code. This makes it possible for different events to be handled by the same callback function, if desired.

Warning: Event set filtering allows false positives for events that have the same layer and layer code. A callback handler function **must** check the event code (passed as an argument) against the specific network events it will handle, **regardless** of how many events were in the set passed to *net_mgmt_init_event_callback()*.

Note that in order to receive events from multiple layers, one must have multiple listeners registered, one for each layer being listened. The callback handler function can be shared between different layer events.

(False positives can occur for events which have the same layer and layer code.)

An example follows.

```
/*
 * Set of events to handle.
 * See e.g. include/net/net_event.h for some NET_EVENT_xxx values.
 */
#define EVENT_IFACE_SET (NET_EVENT_IF_xxx | NET_EVENT_IF_yyy)
#define EVENT_IPV4_SET (NET_EVENT_IPV4_xxx | NET_EVENT_IPV4_yyy)

struct net_mgmt_event_callback iface_callback;
struct net_mgmt_event_callback ipv4_callback;

void callback_handler(struct net_mgmt_event_callback *cb,
                      uint32_t mgmt_event,
                      struct net_if *iface)
{
        if (mgmt_event == NET_EVENT_IF_xxx) {
                /* Handle NET_EVENT_IF_xxx */
        } else if (mgmt_event == NET_EVENT_IF_yyy) {
                /* Handle NET_EVENT_IF_yyy */
        } else if (mgmt_event == NET_EVENT_IPV4_xxx) {
                /* Handle NET_EVENT_IPV4_xxx */
        } else if (mgmt_event == NET_EVENT_IPV4_yyy) {
                /* Handle NET_EVENT_IPV4_yyy */
        } else {
                /* Spurious (false positive) invocation. */
        }
}

void register_cb(void)
{
        net_mgmt_init_event_callback(&iface_callback, callback_handler,
                                     EVENT_IFACE_SET);
        net_mgmt_init_event_callback(&ipv4_callback, callback_handler,
                                     EVENT_IPV4_SET);
        net_mgmt_add_event_callback(&iface_callback);
        net_mgmt_add_event_callback(&ipv4_callback);
}
```

See include/net/net_event.h for available generic core events that can be listened to.

Defining a network management procedure You can provide additional management procedures specific to your stack implementation by defining a handler and registering it with an associated mgmt_request code.

Management request code are defined in relevant places depending on the targeted layer or eventually, if l2 is the layer, on the technology as well. For instance, all IP layer management request code will be found in the include/net/net_event.h header file. But in case of an L2 technology, let's say Ethernet, these would be found in include/net/ethernet.h

You define your handler modeled with this signature:

```
static int your_handler(uint32_t mgmt_event, struct net_if *iface,
                        void *data, size_t len);
```

and then register it with an associated mgmt_request code:

```
NET_MGMT_REGISTER_REQUEST_HANDLER(<mgmt_request code>, your_handler);
```

This new management procedure could then be called by using:

```
net_mgmt(<mgmt_request code>, ...);
```

Signaling a network event You can signal a specific network event using the `net_mgmt_notify()` function and provide the network event code. See include/net/net_mgmt.h for details. As for the management request code, event code can be also found on specific L2 technology mgmt headers, for example include/net/ieee802154_mgmt.h would be the right place if 802.15.4 L2 is the technology one wants to listen to events.

API Reference

group `net_mgmt`

Network Management.

Defines

`net_mgmt(_mgmt_request, _iface, _data, _len)`

`NET_MGMT_DEFINE_REQUEST_HANDLER(_mgmt_request)`

`NET_MGMT_REGISTER_REQUEST_HANDLER(_mgmt_request, _func)`

Typedefs

typedef int (*`net_mgmt_request_handler_t`)(uint32_t mgmt_request, struct *net_if* *iface, void *data, size_t len)

Signature which all Net MGMT request handler need to follow.

Param mgmt_request The exact request value the handler is being called through

Param iface A valid pointer on struct *net_if* if the request is meant to be tight to a network interface. NULL otherwise.

Param data A valid pointer on a data understood by the handler. NULL otherwise.

Param len Length in byte of the memory pointed by data.

typedef void (*`net_mgmt_event_handler_t`)(struct *net_mgmt_event_callback* *cb, uint32_t mgmt_event, struct *net_if* *iface)

Define the user's callback handler function signature.

Param cb Original struct *net_mgmt_event_callback* owning this handler.

Param mgmt_event The network event being notified.

Param iface A pointer on a struct *net_if* to which the the event belongs to, if it's an event on an iface. NULL otherwise.

Functions

static inline void `net_mgmt_init_event_callback`(struct *net_mgmt_event_callback* *cb, *net_mgmt_event_handler_t* handler, uint32_t mgmt_event_mask)

Helper to initialize a struct *net_mgmt_event_callback* properly.

Parameters

- `cb` – A valid application's callback structure pointer.
- `handler` – A valid handler function pointer.
- `mgmt_event_mask` – A mask of relevant events for the handler

void `net_mgmt_add_event_callback`(struct *net_mgmt_event_callback* *cb)

Add a user callback.

Parameters

- `cb` – A valid pointer on user's callback to add.

void `net_mgmt_del_event_callback`(struct *net_mgmt_event_callback* *cb)

Delete a user callback.

Parameters

- `cb` – A valid pointer on user's callback to delete.

void `net_mgmt_event_notify_with_info`(uint32_t mgmt_event, struct *net_if* *iface, const void *info, size_t length)

Used by the system to notify an event.

Note: info and length are disabled if CONFIG_NET_MGMT_EVENT_INFO is not defined.

Parameters

- `mgmt_event` – The actual network event code to notify
- `iface` – a valid pointer on a struct *net_if* if only the event is based on an iface. NULL otherwise.
- `info` – a valid pointer on the information you want to pass along with the event. NULL otherwise. Note the data pointed there is normalized by the related event.
- `length` – size of the data pointed by info pointer.

static inline void `net_mgmt_event_notify`(uint32_t mgmt_event, struct *net_if* *iface)

int `net_mgmt_event_wait`(uint32_t mgmt_event_mask, uint32_t *raised_event, struct *net_if* **iface, const void **info, size_t *info_length, *k_timeout_t* timeout)

Used to wait synchronously on an event mask.

Parameters

- `mgmt_event_mask` – A mask of relevant events to wait on.
- `raised_event` – a pointer on a uint32_t to get which event from the mask generated the event. Can be NULL if the caller is not interested in that information.
- `iface` – a pointer on a place holder for the iface on which the event has originated from. This is valid if only the event mask has bit NET_MGMT_IFACE_BIT set relevantly, depending on events the caller wants to listen to.
- `info` – a valid pointer if user wants to get the information the event might bring along. NULL otherwise.
- `info_length` – tells how long the info memory area is. Only valid if the info is not NULL.
- `timeout` – A timeout delay. K_FOREVER can be used to wait indefinitely.

Returns 0 on success, a negative error code otherwise. -ETIMEDOUT will be specifically returned if the timeout kick-in instead of an actual event.

int net_mgmt_event_wait_on_iface(struct *net_if* *iface, uint32_t mgmt_event_mask, uint32_t *raised_event, const void **info, size_t *info_length, *k_timeout_t* timeout)

Used to wait synchronously on an event mask for a specific iface.

Parameters

- iface – a pointer on a valid network interface to listen event to
- mgmt_event_mask – A mask of relevant events to wait on. Listened to events should be relevant to iface events and thus have the bit NET_MGMT_IFACE_BIT set.
- raised_event – a pointer on a uint32_t to get which event from the mask generated the event. Can be NULL if the caller is not interested in that information.
- info – a valid pointer if user wants to get the information the event might bring along. NULL otherwise.
- info_length – tells how long the info memory area is. Only valid if the info is not NULL.
- timeout – A timeout delay. K_FOREVER can be used to wait indefinitely.

Returns 0 on success, a negative error code otherwise. -ETIMEDOUT will be specifically returned if the timeout kick-in instead of an actual event.

void net_mgmt_event_init(void)

Used by the core of the network stack to initialize the network event processing.

struct net_mgmt_event_callback

#include <net_mgmt.h> Network Management event callback structure Used to register a callback into the network management event part, in order to let the owner of this struct to get network event notification based on given event mask.

Public Members

sys_snode_t node

Meant to be used internally, to insert the callback into a list. So nobody should mess with it.

net_mgmt_event_handler_t handler

Actual callback function being used to notify the owner

struct k_sem *sync_call

Semaphore meant to be used internaly for the synchronous *net_mgmt_event_wait()* function.

uint32_t event_mask

A mask of network events on which the above handler should be called in case those events come. Note that only the command part is treated as a mask, matching one to several commands. Layer and layer code will be made of an exact match. This means that in order to receive events from multiple layers, one must have multiple listeners registered, one for each layer being listened.

uint32_t raised_event

Internal place holder when a synchronous event wait is successfully unlocked on a event.

union *net_mgmt_event_callback*.[anonymous] [anonymous]

A mask of network events on which the above handler should be called in case those events come. Such mask can be modified whenever necessary by the owner, and thus will affect the handler being called or not.

Network Statistics

Overview Network statistics are collected if `CONFIG_NET_STATISTICS` is set. Individual component statistics for IPv4 or IPv6 can be turned off if those statistics are not needed. See various options in subsys/net/ip/Kconfig.stats file for details.

By default, the system collects network statistics per network interface. This can be controlled by `CONFIG_NET_STATISTICS_PER_INTERFACE` option.

The `CONFIG_NET_STATISTICS_USER_API` option can be set if the application wants to collect statistics for further processing. The network management interface API is used for that. See *Network Management* for details.

The `CONFIG_NET_STATISTICS_ETHERNET` option can be set to collect generic Ethernet statistics. If the `CONFIG_NET_STATISTICS_ETHERNET_VENDOR` option is set, then Ethernet device driver can collect Ethernet device specific statistics. These statistics can then be transferred to application for processing.

If the `CONFIG_NET_SHELL` option is set, then network shell can show statistics information with `net stats` command.

API Reference

group `net_stats`

Network statistics library.

Defines

`NET_TC_TX_STATS_COUNT`

`NET_TC_RX_STATS_COUNT`

Typedefs

typedef uint32_t `net_stats_t`

Network statistics counter.

struct `net_stats_bytes`

#include <net_stats.h> Number of bytes sent and received.

Public Members

net_stats_t `sent`
Number of bytes sent

net_stats_t `received`
Number of bytes received

struct `net_stats_pkts`
#include <net_stats.h> Number of network packets sent and received.

Public Members

net_stats_t `tx`
Number of packets sent

net_stats_t `rx`
Number of packets received

struct `net_stats_ip`
#include <net_stats.h> IP layer statistics.

Public Members

net_stats_t `recv`
Number of received packets at the IP layer.

net_stats_t `sent`
Number of sent packets at the IP layer.

net_stats_t `forwarded`
Number of forwarded packets at the IP layer.

net_stats_t `drop`
Number of dropped packets at the IP layer.

struct `net_stats_ip_errors`
#include <net_stats.h> IP layer error statistics.

Public Members

net_stats_t `vhlerr`
Number of packets dropped due to wrong IP version or header length.

net_stats_t `hblenerr`
Number of packets dropped due to wrong IP length, high byte.

net_stats_t `lblenerr`
Number of packets dropped due to wrong IP length, low byte.

net_stats_t `fragerr`
Number of packets dropped because they were IP fragments.

net_stats_t `chkerr`
Number of packets dropped due to IP checksum errors.

net_stats_t `protoerr`
Number of packets dropped because they were neither ICMP, UDP nor TCP.

struct `net_stats_icmp`
#include <net_stats.h> ICMP statistics.

Public Members

net_stats_t `recv`
Number of received ICMP packets.

net_stats_t `sent`
Number of sent ICMP packets.

net_stats_t `drop`
Number of dropped ICMP packets.

net_stats_t `typeerr`
Number of ICMP packets with a wrong type.

net_stats_t `chkerr`
Number of ICMP packets with a bad checksum.

struct `net_stats_tcp`
#include <net_stats.h> TCP statistics.

Public Members

struct *net_stats_bytes* `bytes`
Amount of received and sent TCP application data.

net_stats_t `resent`
Amount of retransmitted data.

net_stats_t `drop`
Number of dropped packets at the TCP layer.

net_stats_t `recv`
Number of received TCP segments.

net_stats_t `sent`
Number of sent TCP segments.

net_stats_t `seg_drop`
Number of dropped TCP segments.

net_stats_t `chkerr`
Number of TCP segments with a bad checksum.

net_stats_t `ackerr`
Number of received TCP segments with a bad ACK number.

net_stats_t `rsterr`
Number of received bad TCP RST (reset) segments.

net_stats_t `rst`
Number of received TCP RST (reset) segments.

net_stats_t `rexmit`
Number of retransmitted TCP segments.

net_stats_t `conndrop`
Number of dropped connection attempts because too few connections were available.

net_stats_t `connrst`
Number of connection attempts for closed ports, triggering a RST.

struct `net_stats_udp`
#include <net_stats.h> UDP statistics.

Public Members

net_stats_t `drop`
Number of dropped UDP segments.

net_stats_t `recv`
Number of received UDP segments.

net_stats_t `sent`
Number of sent UDP segments.

net_stats_t `chkerr`
Number of UDP segments with a bad checksum.

struct net_stats_ipv6_nd

#include <net_stats.h> IPv6 neighbor discovery statistics.

struct net_stats_ipv6_mld

#include <net_stats.h> IPv6 multicast listener daemon statistics.

Public Members

net_stats_t recv

Number of received IPv6 MLD queries

net_stats_t sent

Number of sent IPv6 MLD reports

net_stats_t drop

Number of dropped IPv6 MLD packets

struct net_stats_ipv4_igmp

#include <net_stats.h> IPv4 IGMP daemon statistics.

Public Members

net_stats_t recv

Number of received IPv4 IGMP queries

net_stats_t sent

Number of sent IPv4 IGMP reports

net_stats_t drop

Number of dropped IPv4 IGMP packets

struct net_stats_tx_time

#include <net_stats.h> Network packet transfer times for calculating average TX time.

struct net_stats_rx_time

#include <net_stats.h> Network packet receive times for calculating average RX time.

struct net_stats_tc

#include <net_stats.h> Traffic class statistics.

struct net_stats_pm

#include <net_stats.h> Power management statistics.

struct net_stats

#include <net_stats.h> All network statistics in one struct.

Public Members

net_stats_t `processing_error`
Count of malformed packets or packets we do not have handler for

struct *net_stats_bytes* `bytes`
This calculates amount of data transferred through all the network interfaces.

struct *net_stats_ip_errors* `ip_errors`
IP layer errors

struct `net_stats_eth_errors`
#include <net_stats.h> Ethernet error statistics.

struct `net_stats_eth_flow`
#include <net_stats.h> Ethernet flow control statistics.

struct `net_stats_eth_csum`
#include <net_stats.h> Ethernet checksum statistics.

struct `net_stats_eth_hw_timestamp`
#include <net_stats.h> Ethernet hardware timestamp statistics.

struct `net_stats_eth`
#include <net_stats.h> All Ethernet specific statistics.

struct `net_stats_ppp`
#include <net_stats.h> All PPP specific statistics.

Public Members

net_stats_t `drop`
Number of received and dropped PPP frames.

net_stats_t `chkerr`
Number of received PPP frames with a bad checksum.

Network Timeout

- *Overview*
- *Use*
- *API Reference*

Overview Zephyr's network infrastructure mostly uses the millisecond-resolution uptime clock to track timeouts, with both deadlines and durations measured with 32-bit unsigned values. The 32-bit value rolls over at 49 days 17 hours 2 minutes 47.296 seconds.

Timeout processing is often affected by latency, so that the time at which the timeout is checked may be some time after it should have expired. Handling this correctly without arbitrary expectations of maximum latency requires that the maximum delay that can be directly represented be a 31-bit non-negative number (`INT32_MAX`), which overflows at 24 days 20 hours 31 minutes 23.648 seconds.

Most network timeouts are shorter than the delay rollover, but a few protocols allow for delays that are represented as unsigned 32-bit values counting seconds, which corresponds to a 42-bit millisecond count.

The net_timeout API provides a generic timeout mechanism to correctly track the remaining time for these extended-duration timeouts.

Use The simplest use of this API is:

1. Configure a network timeout using *net_timeout_set()*.
2. Use *net_timeout_evaluate()* to determine how long it is until the timeout occurs. Schedule a timeout to occur after this delay.
3. When the timeout callback is invoked, use *net_timeout_evaluate()* again to determine whether the timeout has completed, or whether there is additional time remaining. If the latter, reschedule the callback.
4. While the timeout is running, use *net_timeout_remaining()* to get the number of seconds until the timeout expires. This may be used to explicitly update the timeout, which should be done by canceling any pending callback and restarting from step 1 with the new timeout.

The *net_timeout* contains a `sys_snode_t` that allows multiple timeout instances to be aggregated to share a single kernel timer element. The application must use *net_timeout_evaluate()* on all instances to determine the next timeout event to occur.

net_timeout_deadline() may be used to reconstruct the full-precision deadline of the timeout. This exists primarily for testing but may have use in some applications, as it does allow a millisecond-resolution calculation of remaining time.

API Reference

group `net_timeout`

Network long timeout primitives and helpers.

Defines

`NET_TIMEOUT_MAX_VALUE`

Divisor used to support ms resolution timeouts.

Because delays are processed in work queues which are not invoked synchronously with clock changes we need to be able to detect timeouts after they occur, which requires comparing "deadline" to "now" with enough "slop" to handle any observable latency due to "now" advancing past "deadline".

The simplest solution is to use the native conversion of the well-defined 32-bit unsigned difference to a 32-bit signed difference, which caps the maximum delay at INT32_MAX. This is compatible with the standard mechanism for detecting completion of deadlines that do not overflow their representation.

Functions

void `net_timeout_set`(struct *net_timeout* *timeout, uint32_t lifetime, uint32_t now)

Configure a network timeout structure.

Parameters

- `timeout` – a pointer to the timeout state.
- `lifetime` – the duration of the timeout in seconds.
- `now` – the time at which the timeout started counting down, in milliseconds. This is generally a captured value of *k_uptime_get_32()*.

int64_t `net_timeout_deadline`(const struct *net_timeout* *timeout, int64_t now)

Return the 64-bit system time at which the timeout will complete.

Note: Correct behavior requires invocation of *net_timeout_evaluate()* at its specified intervals.

Parameters

- `timeout` – state a pointer to the timeout state, initialized by *net_timeout_set()* and maintained by *net_timeout_evaluate()*.
- `now` – the full-precision value of *k_uptime_get()* relative to which the deadline will be calculated.

Returns the value of *k_uptime_get()* at which the timeout will expire.

uint32_t `net_timeout_remaining`(const struct *net_timeout* *timeout, uint32_t now)

Calculate the remaining time to the timeout in whole seconds.

Note: This function rounds the remaining time down, i.e. if the timeout will occur in 3500 milliseconds the value 3 will be returned.

Note: Correct behavior requires invocation of *net_timeout_evaluate()* at its specified intervals.

Parameters

- `timeout` – a pointer to the timeout state
- `now` – the time relative to which the estimate of remaining time should be calculated. This should be recently captured value from *k_uptime_get_32()*.

Return values

- `0` – if the timeout has completed.
- `positive` – the remaining duration of the timeout, in seconds.

uint32_t `net_timeout_evaluate`(struct *net_timeout* *timeout, uint32_t now)

Update state to reflect elapsed time and get new delay.

This function must be invoked periodically to (1) apply the effect of elapsed time on what remains of a total delay that exceeded the maximum representable delay, and (2) determine that either the timeout has completed or that the infrastructure must wait a certain period before checking again for completion.

Parameters

- `timeout` – a pointer to the timeout state

- `now` – the time relative to which the estimate of remaining time should be calculated. This should be recently captured value from *k_uptime_get_32()*.

Return values

- `0` – if the timeout has completed
- `positive` – the maximum delay until the state of this timeout should be re-evaluated, in milliseconds.

struct `net_timeout`

#include <net_timeout.h> Generic struct for handling network timeouts.

Except for the linking node, all access to state from these objects must go through the defined API.

Public Members

sys_snode_t `node`

Used to link multiple timeouts that share a common timer infrastructure.

For examples a set of related timers may use a single delayed work structure, which is always scheduled at the shortest time to a timeout event.

Networking Context

The net_context API is not meant for application use. Application should use *BSD Sockets* API instead.

Promiscuous Mode

- *Overview*
- *Sample usage*
- *API Reference*

Overview Promiscuous mode is a mode for a network interface controller that causes it to pass all traffic it receives to the application rather than passing only the frames that the controller is specifically programmed to receive. This mode is normally used for packet sniffing as used to diagnose network connectivity issues by showing an application all the data being transferred over the network. (See the Wikipedia article on promiscuous mode for more information.)

The network promiscuous APIs are used to enable and disable this mode, and to wait for and receive a network data to arrive. Not all network technologies or network device drivers support promiscuous mode.

Sample usage First the promiscuous mode needs to be turned ON by the application like this:

```
ret = net_promisc_mode_on(iface);
if (ret < 0) {
        if (ret == -EALREADY) {
                printf("Promiscuous mode already enabled\n");
        } else {
                printf("Cannot enable promiscuous mode for "
```

(continues on next page)

(continued from previous page)

```
                        "interface %p (%d)\n", iface, ret);
        }
}
```

If there is no error, then the application can start to wait for network data:

```
while (true) {
        pkt = net_promisc_mode_wait_data(K_FOREVER);
        if (pkt) {
                print_info(pkt);
        }

        net_pkt_unref(pkt);
}
```

Finally the promiscuous mode can be turned OFF by the application like this:

```
ret = net_promisc_mode_off(iface);
if (ret < 0) {
        if (ret == -EALREADY) {
                printf("Promiscuous mode already disabled\n");
        } else {
                printf("Cannot disable promiscuous mode for "
                       "interface %p (%d)\n", iface, ret);
        }
}
```

See net-promiscuous-mode-sample for a more comprehensive example.

API Reference

group `promiscuous`

Promiscuous mode support.

Functions

static inline struct *net_pkt* *`net_promisc_mode_wait_data`(*k_timeout_t* timeout)

Start to wait received network packets.

Parameters

- `timeout` – How long to wait before returning.

Returns Received *net_pkt*, NULL if not received any packet.

static inline int `net_promisc_mode_on`(struct *net_if* *iface)

Enable promiscuous mode for a given network interface.

Parameters

- `iface` – Network interface

Returns 0 if ok, <0 if error

static inline int `net_promisc_mode_off`(struct *net_if* *iface)

Disable promiscuous mode for a given network interface.

Parameters

- `iface` – Network interface

Returns 0 if ok, <0 if error

Simple Network Time Protocol Library

Overview The SNTP library implements IETF RFC4330 (Simple Network Time Protocol v4).

SNTP provides a way to synchronize clocks in computer networks.

API Reference

group `sntp`

Simple Network Time Protocol API.

Functions

int `sntp_init`(struct *sntp_ctx* *ctx, struct *sockaddr* *addr, *socklen_t* addr_len)

Initialize SNTP context.

Parameters

- `ctx` – Address of sntp context.
- `addr` – IP address of NTP/SNTP server.
- `addr_len` – IP address length of NTP/SNTP server.

Returns 0 if ok, <0 if error.

int `sntp_query`(struct *sntp_ctx* *ctx, uint32_t timeout, struct *sntp_time* *time)

Perform SNTP query.

Parameters

- `ctx` – Address of sntp context.
- `timeout` – Timeout of waiting for sntp response (in milliseconds).
- `time` – Timestamp including integer and fractional seconds since 1 Jan 1970 (output).

Returns 0 if ok, <0 if error (-ETIMEDOUT if timeout).

void `sntp_close`(struct *sntp_ctx* *ctx)

Release SNTP context.

Parameters

- `ctx` – Address of sntp context.

int `sntp_simple`(const char *server, uint32_t timeout, struct *sntp_time* *time)

Convenience function to query SNTP in one-shot fashion.

Convenience wrapper which calls getaddrinfo(), *sntp_init()*, *sntp_query()*, and *sntp_close()*.

Parameters

- `server` – Address of server in format addr[:port]

- `timeout` – Query timeout
- `time` – Timestamp including integer and fractional seconds since 1 Jan 1970 (output).

Returns 0 if ok, <0 if error (-ETIMEDOUT if timeout).

struct `sntp_ctx`

#include <sntp.h> SNTP context

Public Members

uint32_t `expected_orig_ts`

Timestamp when the request was sent from client to server. This is used to check if the originated timestamp in the server reply matches the one in client request.

struct `sntp_time`

#include <sntp.h> Time as returned by SNTP API, fractional seconds since 1 Jan 1970

SOCKS5 Proxy Support

Overview The SOCKS library implements SOCKS5 support, which allows Zephyr to connect to peer devices via a network proxy.

See this SOCKS5 Wikipedia article for a detailed overview of how SOCKS5 works.

For more information about the protocol itself, see IETF RFC1928 SOCKS Protocol Version 5.

SOCKS5 API The SOCKS5 support is enabled by `CONFIG_SOCKS` Kconfig variable. Application wanting to use the SOCKS5 must set the SOCKS5 proxy host adddress by calling `setsockopt()` like this:

```
static int set_proxy(int sock, const struct sockaddr *proxy_addr,
                     socklen_t proxy_addrlen)
{
    int ret;

    ret = setsockopt(sock, SOL_SOCKET, SO_SOCKS5,
                     proxy_addr, proxy_addrlen);
    if (ret < 0) {
            return -errno;
    }

    return 0;
}
```

SOCKS5 Proxy Usage in MQTT For MQTT client, there is `mqtt_client_set_proxy()` API that the application can call to setup SOCKS5 proxy. See mqtt-publisher-sample for usage example.

Trickle Timer Library

Overview The Trickle timer library implements IETF RFC6206 (Trickle Algorithm).

The Trickle algorithm allows nodes in a lossy shared medium (e.g., low-power and lossy networks) to exchange information in a highly robust, energy efficient, simple, and scalable manner.

API Reference

group `trickle`

Trickle algorithm library.

Typedefs

typedef void (*`net_trickle_cb_t`)(struct *net_trickle* *trickle, bool do_suppress, void *user_data)

Trickle timer callback.

The callback is called after Trickle timeout expires.

Param trickle The trickle context to use.

Param do_suppress Is TX allowed (true) or not (false).

Param user_data The user data given in *net_trickle_start()* call.

Functions

int `net_trickle_create`(struct *net_trickle* *trickle, uint32_t Imin, uint8_t Imax, uint8_t k)

Create a Trickle timer.

Parameters

- `trickle` – Pointer to Trickle struct.
- `Imin` – Imin configuration parameter in ms.
- `Imax` – Max number of doublings.
- `k` – Redundancy constant parameter. See RFC 6206 for details.

Returns Return 0 if ok and <0 if error.

int `net_trickle_start`(struct *net_trickle* *trickle, *net_trickle_cb_t* cb, void *user_data)

Start a Trickle timer.

Parameters

- `trickle` – Pointer to Trickle struct.
- `cb` – User callback to call at time T within the current trickle interval
- `user_data` – User pointer that is passed to callback.

Returns Return 0 if ok and <0 if error.

int net_trickle_stop(struct *net_trickle* *trickle)

Stop a Trickle timer.

Parameters

- trickle – Pointer to Trickle struct.

Returns Return 0 if ok and <0 if error.

void net_trickle_consistency(struct *net_trickle* *trickle)

To be called by the protocol handler when it hears a consistent network transmission.

Parameters

- trickle – Pointer to Trickle struct.

void net_trickle_inconsistency(struct *net_trickle* *trickle)

To be called by the protocol handler when it hears an inconsistent network transmission.

Parameters

- trickle – Pointer to Trickle struct.

static inline bool net_trickle_is_running(struct *net_trickle* *trickle)

Check if the Trickle timer is running or not.

Parameters

- trickle – Pointer to Trickle struct.

Returns Return True if timer is running and False if not.

struct net_trickle

#include <trickle.h> The variable names are taken directly from RFC 6206 when applicable. Note that the struct members should not be accessed directly but only via the Trickle API.

Public Members

uint32_t Imin

Min interval size in ms

uint8_t Imax

Max number of doublings

uint8_t k

Redundancy constant

uint32_t I

Current interval size

uint32_t Istart

Start of the interval in ms

uint8_t c

Consistency counter

uint32_t Imax_abs

Max interval size in ms (not doublings)

net_trickle_cb_t `cb`

Callback to be called when timer expires

Websocket Client API

Overview The Websocket client library allows Zephyr to connect to a Websocket server. The Websocket client API can be used directly by application to establish a Websocket connection to server, or it can be used as a transport for other network protocols like MQTT.

See this Websocket Wikipedia article for a detailed overview of how Websocket works.

For more information about the protocol itself, see IETF RFC6455 The WebSocket Protocol.

Websocket Transport The Websocket API allows it to be used as a transport for other high level protocols like MQTT. The Zephyr MQTT client library can be configured to use Websocket transport by enabling `CONFIG_MQTT_LIB_WEBSOCKET` and `CONFIG_WEBSOCKET_CLIENT` Kconfig options.

First a socket needs to be created and connected to the Websocket server:

```
sock = socket(family, SOCK_STREAM, IPPROTO_TCP);
...
ret = connect(sock, addr, addr_len);
...
```

The Websocket transport socket is then created like this:

```
ws_sock = websocket_connect(sock, &config, timeout, user_data);
```

The Websocket socket can then be used to send or receive data, and the Websocket client API will encapsulate the sent or received data to/from Websocket packet payload. Both the `websocket_xxx()` API or normal BSD socket API functions can be used to send and receive application data.

```
ret = websocket_send_msg(ws_sock, buf_to_send, buf_len,
                         WEBSOCKET_OPCODE_DATA_BINARY, true, true,
                         K_FOREVER);
...
ret = send(ws_sock, buf_to_send, buf_len, 0);
```

If normal BSD socket functions are used, then currently only TEXT data is supported. In order to send BINARY data, the *websocket_send_msg()* must be used.

When done, the Websocket transport socket must be closed.

```
ret = close(ws_sock);
or
ret = websocket_disconnect(ws_sock);
```

API Reference

group `websocket`

Websocket API.

Defines

`WEBSOCKET_FLAG_FINAL`

Message type values. Returned in *websocket_recv_msg()* Final frame

`WEBSOCKET_FLAG_TEXT`

Textual data

`WEBSOCKET_FLAG_BINARY`

Binary data

`WEBSOCKET_FLAG_CLOSE`

Closing connection

`WEBSOCKET_FLAG_PING`

Ping message

`WEBSOCKET_FLAG_PONG`

Pong message

Typedefs

typedef int (*`websocket_connect_cb_t`)(int ws_sock, struct http_request *req, void *user_data)

Callback called after Websocket connection is established.

Param ws_sock Websocket id

Param req HTTP handshake request

Param user_data A valid pointer on some user data or NULL

Return 0 if ok, <0 if there is an error and connection should be aborted

Enums

enum `websocket_opcode`

Values:

enumerator `WEBSOCKET_OPCODE_CONTINUE` = 0x00

enumerator `WEBSOCKET_OPCODE_DATA_TEXT` = 0x01

enumerator `WEBSOCKET_OPCODE_DATA_BINARY` = 0x02

enumerator WEBSOCKET_OPCODE_CLOSE = 0x08

enumerator WEBSOCKET_OPCODE_PING = 0x09

enumerator WEBSOCKET_OPCODE_PONG = 0x0A

Functions

int websocket_connect(int http_sock, struct *websocket_request* *req, int32_t timeout, void *user_data)

Connect to a server that provides Websocket service. The callback is called after connection is established. The returned value is a new socket descriptor that can be used to send / receive data using the BSD socket API.

Parameters

- http_sock – Socket id to the server. Note that this socket is used to do HTTP handshakes etc. The actual Websocket connectivity is done via the returned websocket id. Note that the http_sock must not be closed after this function returns as it is used to deliver the Websocket packets to the Websocket server.
- req – Websocket request. User should allocate and fill the request data.
- timeout – Max timeout to wait for the connection. The timeout value is in milliseconds. Value SYS_FOREVER_MS means to wait forever.
- user_data – User specified data that is passed to the callback.

Returns Websocket id to be used when sending/receiving Websocket data.

int websocket_send_msg(int ws_sock, const uint8_t *payload, size_t payload_len, enum *websocket_opcode* opcode, bool mask, bool final, int32_t timeout)

Send websocket msg to peer.

The function will automatically add websocket header to the message.

Parameters

- ws_sock – Websocket id returned by *websocket_connect()*.
- payload – Websocket data to send.
- payload_len – Length of the data to be sent.
- opcode – Operation code (text, binary, ping, pong, close)
- mask – Mask the data, see RFC 6455 for details
- final – Is this final message for this message send. If final == false, then the first message must have valid opcode and subsequent messages must have opcode WEBSOCKET_OPCODE_CONTINUE. If final == true and this is the only message, then opcode should have proper opcode (text or binary) set.
- timeout – How long to try to send the message. The value is in milliseconds. Value SYS_FOREVER_MS means to wait forever.

Returns <0 if error, >=0 amount of bytes sent

int websocket_recv_msg(int ws_sock, uint8_t *buf, size_t buf_len, uint32_t *message_type, uint64_t *remaining, int32_t timeout)

Receive websocket msg from peer.

The function will automatically remove websocket header from the message.

Parameters

- `ws_sock` – Websocket id returned by *websocket_connect()*.
- `buf` – Buffer where websocket data is read.
- `buf_len` – Length of the data buffer.
- `message_type` – Type of the message.
- `remaining` – How much there is data left in the message after this read.
- `timeout` – How long to try to receive the message. The value is in milliseconds. Value SYS_FOREVER_MS means to wait forever.

Returns <0 if error, >=0 amount of bytes received

int `websocket_disconnect`(int ws_sock)

Close websocket.

One must call *websocket_connect()* after this call to re-establish the connection.

Parameters

- `ws_sock` – Websocket id returned by *websocket_connect()*.

static inline void `websocket_init`(void)

struct `websocket_request`

#include <websocket.h> Websocket client connection request. This contains all the data that is needed when doing a Websocket connection request.

Public Members

const char *`host`

Host of the Websocket server when doing HTTP handshakes.

const char *`url`

URL of the Websocket.

http_header_cb_t `optional_headers_cb`

User supplied callback function to call when optional headers need to be sent. This can be NULL, in which case the optional_headers field in http_request is used. The idea of this optional_headers callback is to allow user to send more HTTP header data that is practical to store in allocated memory.

const char **`optional_headers`

A NULL terminated list of any optional headers that should be added to the HTTP request. May be NULL. If the optional_headers_cb is specified, then this field is ignored.

websocket_connect_cb_t `cb`

User supplied callback function to call when a connection is established.

const struct http_parser_settings *`http_cb`

User supplied list of callback functions if the calling application wants to know the parsing status or the HTTP fields during the handshake. This is optional parameter and normally not needed but is useful if the caller wants to know something about the fields that the server is sending.

uint8_t *tmp_buf

User supplied buffer where HTTP connection data is stored

size_t tmp_buf_len

Length of the user supplied temp buffer

Network Packet Capture

Overview The net_capture API allows user to monitor the network traffic in one of the Zephyr network interfaces and send that traffic to external system for analysis. The monitoring can be setup either manually using net-shell or automatically by using the net_capture API.

Sample usage See Network capture sample application and *Monitor Network Traffic* for details.

API Reference

group net_capture

Network packet capture support functions.

Functions

int net_capture_setup(const char *remote_addr, const char *my_local_addr, const char *peer_addr, const struct *device* **dev)

Setup network packet capturing support.

Parameters

- remote_addr – The value tells the tunnel remote/outer endpoint IP address. The IP address can be either IPv4 or IPv6 address. This address is used to select the network interface where the tunnel is created.
- my_local_addr – The local/inner IP address of the tunnel. Can contain also port number which is used as UDP source port.
- peer_addr – The peer/inner IP address of the tunnel. Can contain also port number which is used as UDP destination port.
- dev – Network capture device. This is returned to the caller.

Returns 0 if ok, <0 if network packet capture setup failed

static inline int net_capture_cleanup(const struct *device* *dev)

Cleanup network packet capturing support.

This should be called after the capturing is done and resources can be released.

Parameters

- dev – Network capture device. User must allocate using the *net_capture_setup()* function.

Returns 0 if ok, <0 if network packet capture cleanup failed

static inline int `net_capture_enable`(const struct *device* *dev, struct *net_if* *iface)

Enable network packet capturing support.

This creates tunnel network interface where all the captured packets are pushed. The captured network packets are placed in UDP packets that are sent to tunnel peer.

Parameters

- `dev` – Network capture device
- `iface` – Network interface we are starting to capture packets.

Returns 0 if ok, <0 if network packet capture enable failed

static inline bool `net_capture_is_enabled`(const struct *device* *dev)

Is network packet capture enabled or disabled.

Parameters

- `dev` – Network capture device

Returns True if enabled, False if network capture is disabled.

static inline int `net_capture_disable`(const struct *device* *dev)

Disable network packet capturing support.

Parameters

- `dev` – Network capture device

Returns 0 if ok, <0 if network packet capture disable failed

static inline int `net_capture_send`(const struct *device* *dev, struct *net_if* *iface, struct *net_pkt* *pkt)

Send captured packet.

Parameters

- `dev` – Network capture device
- `iface` – Network interface the packet is being sent
- `pkt` – The network packet that is sent

Returns 0 if ok, <0 if network packet capture send failed

7.20.2 Network Buffer Management

Network Buffer

Overview Network buffers are a core concept of how the networking stack (as well as the Bluetooth stack) pass data around. The API for them is defined in include/net/buf.h:.

Creating buffers Network buffers are created by first defining a pool of them:

```
NET_BUF_POOL_DEFINE(pool_name, buf_count, buf_size, user_data_size, NULL);
```

The pool is a static variable, so if it's needed to be exported to another module a separate pointer is needed.

Once the pool has been defined, buffers can be allocated from it with:

```
buf = net_buf_alloc(&pool_name, timeout);
```

There is no explicit initialization function for the pool or its buffers, rather this is done implicitly as *net_buf_alloc()* gets called.

If there is a need to reserve space in the buffer for protocol headers to be prepended later, it's possible to reserve this headroom with:

```
net_buf_reserve(buf, headroom);
```

In addition to actual protocol data and generic parsing context, network buffers may also contain protocol-specific context, known as user data. Both the maximum data and user data capacity of the buffers is compile-time defined when declaring the buffer pool.

The buffers have native support for being passed through k_fifo kernel objects. This is a very practical feature when the buffers need to be passed from one thread to another. However, since a net_buf may have a fragment chain attached to it, instead of using the *k_fifo_put()* and *k_fifo_get()* APIs, special *net_buf_put()* and *net_buf_get()* APIs must be used when passing buffers through FIFOs. These APIs ensure that the buffer chains stay intact. The same applies for passing buffers through a singly linked list, in which case the *net_buf_slist_put()* and *net_buf_slist_get()* functions must be used instead of *sys_slist_append()* and *sys_slist_get()*.

Common Operations The network buffer API provides some useful helpers for encoding and decoding data in the buffers. To fully understand these helpers it's good to understand the basic names of operations used with them:

Add Add data to the end of the buffer. Modifies the data length value while leaving the actual data pointer intact. Requires that there is enough tailroom in the buffer. Some examples of APIs for adding data:

```
void *net_buf_add(struct net_buf *buf, size_t len);
void *net_buf_add_mem(struct net_buf *buf, const void *mem, size_t len);
uint8_t *net_buf_add_u8(struct net_buf *buf, uint8_t value);
void net_buf_add_le16(struct net_buf *buf, uint16_t value);
void net_buf_add_le32(struct net_buf *buf, uint32_t value);
```

Remove Remove data from the end of the buffer. Modifies the data length value while leaving the actual data pointer intact. Some examples of APIs for removing data:

```
void *net_buf_remove_mem(struct net_buf *buf, size_t len);
uint8_t net_buf_remove_u8(struct net_buf *buf);
uint16_t net_buf_remove_le16(struct net_buf *buf);
uint32_t net_buf_remove_le32(struct net_buf *buf);
```

Push Prepend data to the beginning of the buffer. Modifies both the data length value as well as the data pointer. Requires that there is enough headroom in the buffer. Some examples of APIs for pushing data:

```
void *net_buf_push(struct net_buf *buf, size_t len);
void *net_buf_push_mem(struct net_buf *buf, const void *mem, size_t len);
void net_buf_push_u8(struct net_buf *buf, uint8_t value);
void net_buf_push_le16(struct net_buf *buf, uint16_t value);
```

Pull Remove data from the beginning of the buffer. Modifies both the data length value as well as the data pointer. Some examples of APIs for pulling data:

```
void *net_buf_pull(struct net_buf *buf, size_t len);
void *net_buf_pull_mem(struct net_buf *buf, size_t len);
uint8_t net_buf_pull_u8(struct net_buf *buf);
uint16_t net_buf_pull_le16(struct net_buf *buf);
uint32_t net_buf_pull_le32(struct net_buf *buf);
```

The Add and Push operations are used when encoding data into the buffer, whereas the Remove and Pull operations are used when decoding data from a buffer.

Reference Counting Each network buffer is reference counted. The buffer is initially acquired from a free buffers pool by calling *net_buf_alloc()*, resulting in a buffer with reference count 1. The reference count can be incremented with *net_buf_ref()* or decremented with *net_buf_unref()*. When the count drops to zero the buffer is automatically placed back to the free buffers pool.

API Reference

group net_buf

Network buffer library.

Defines

NET_BUF_SIMPLE_DEFINE(_name, _size)

Define a *net_buf_simple* stack variable.

This is a helper macro which is used to define a *net_buf_simple* object on the stack.

Parameters

- _name – Name of the *net_buf_simple* object.
- _size – Maximum data storage for the buffer.

NET_BUF_SIMPLE_DEFINE_STATIC(_name, _size)

Define a static *net_buf_simple* variable.

This is a helper macro which is used to define a static *net_buf_simple* object.

Parameters

- _name – Name of the *net_buf_simple* object.
- _size – Maximum data storage for the buffer.

NET_BUF_SIMPLE(_size)

Define a *net_buf_simple* stack variable and get a pointer to it.

This is a helper macro which is used to define a *net_buf_simple* object on the stack and the get a pointer to it as follows:

struct *net_buf_simple* *my_buf = *NET_BUF_SIMPLE(10)*;

After creating the object it needs to be initialized by calling *net_buf_simple_init()*.

Parameters

- _size – Maximum data storage for the buffer.

Returns Pointer to stack-allocated *net_buf_simple* object.

NET_BUF_FRAGS

Flag indicating that the buffer has associated fragments. Only used internally by the buffer handling code while the buffer is inside a FIFO, meaning this never needs to be explicitly set or unset by the *net_buf* API user. As long as the buffer is outside of a FIFO, i.e. in practice always for the user for this API, the buf->frags pointer should be used instead.

NET_BUF_EXTERNAL_DATA

Flag indicating that the buffer's associated data pointer, points to externally allocated memory. Therefore once ref goes down to zero, the pointed data will not need to be deallocated. This never needs to be explicitly set or unet by the *net_buf* API user. Such *net_buf* is exclusively instantiated via *net_buf_alloc_with_data()* function. Reference count mechanism however will behave the same way, and ref count going to 0 will free the *net_buf* but no the data pointer in it.

NET_BUF_POOL_HEAP_DEFINE(_name, _count, _destroy)

Define a new pool for buffers using the heap for the data.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this, the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

The data payload of the buffers will be allocated from the heap using k_malloc, so CONFIG_HEAP_MEM_POOL_SIZE must be set to a positive value. This kind of pool does not support blocking on the data allocation, so the timeout passed to net_buf_alloc will be always treated as K_NO_WAIT when trying to allocate the data. This means that allocation failures, i.e. NULL returns, must always be handled cleanly.

If provided with a custom destroy callback, this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- `_name` – Name of the pool variable.
- `_count` – Number of buffers in the pool.
- `_destroy` – Optional destroy callback when buffer is freed.

NET_BUF_POOL_FIXED_DEFINE(_name, _count, _data_size, _destroy)

Define a new pool for buffers based on fixed-size data.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this, the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

The data payload of the buffers will be allocated from a byte array of fixed sized chunks. This kind of pool does not support blocking on the data allocation, so the timeout passed to net_buf_alloc will be always treated as K_NO_WAIT when trying to allocate the data. This means that allocation failures, i.e. NULL returns, must always be handled cleanly.

If provided with a custom destroy callback, this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- `_name` – Name of the pool variable.
- `_count` – Number of buffers in the pool.
- `_data_size` – Maximum data payload per buffer.

- `_destroy` – Optional destroy callback when buffer is freed.

`NET_BUF_POOL_VAR_DEFINE(_name, _count, _data_size, _destroy)`

Define a new pool for buffers with variable size payloads.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this, the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

The data payload of the buffers will be based on a memory pool from which variable size payloads may be allocated.

If provided with a custom destroy callback, this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- `_name` – Name of the pool variable.
- `_count` – Number of buffers in the pool.
- `_data_size` – Total amount of memory available for data payloads.
- `_destroy` – Optional destroy callback when buffer is freed.

`NET_BUF_POOL_DEFINE(_name, _count, _size, _ud_size, _destroy)`

Define a new pool for buffers.

Defines a *net_buf_pool* struct and the necessary memory storage (array of structs) for the needed amount of buffers. After this,the buffers can be accessed from the pool through net_buf_alloc. The pool is defined as a static variable, so if it needs to be exported outside the current module this needs to happen with the help of a separate pointer rather than an extern declaration.

If provided with a custom destroy callback this callback is responsible for eventually calling *net_buf_destroy()* to complete the process of returning the buffer to the pool.

Parameters

- `_name` – Name of the pool variable.
- `_count` – Number of buffers in the pool.
- `_size` – Maximum data size for each buffer.
- `_ud_size` – Amount of user data space to reserve.
- `_destroy` – Optional destroy callback when buffer is freed.

Typedefs

typedef struct *net_buf* *(*`net_buf_allocator_cb`)(*k_timeout_t* timeout, void *user_data)

Network buffer allocator callback.

The allocator callback is called when net_buf_append_bytes needs to allocate a new *net_buf*.

Param timeout Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Param user_data The user data given in net_buf_append_bytes call.

Return pointer to allocated *net_buf* or NULL on error.

Functions

static inline void `net_buf_simple_init`(struct *net_buf_simple* *buf, size_t reserve_head)

Initialize a *net_buf_simple* object.

This needs to be called after creating a *net_buf_simple* object using the NET_BUF_SIMPLE macro.

Parameters

- `buf` – Buffer to initialize.
- `reserve_head` – Headroom to reserve.

void `net_buf_simple_init_with_data`(struct *net_buf_simple* *buf, void *data, size_t size)

Initialize a *net_buf_simple* object with data.

Initialized buffer object with external data.

Parameters

- `buf` – Buffer to initialize.
- `data` – External data pointer
- `size` – Amount of data the pointed data buffer if able to fit.

static inline void `net_buf_simple_reset`(struct *net_buf_simple* *buf)

Reset buffer.

Reset buffer data so it can be reused for other purposes.

Parameters

- `buf` – Buffer to reset.

void `net_buf_simple_clone`(const struct *net_buf_simple* *original, struct *net_buf_simple* *clone)

Clone buffer state, using the same data buffer.

Initializes a buffer to point to the same data as an existing buffer. Allows operations on the same data without altering the length and offset of the original.

Parameters

- `original` – Buffer to clone.
- `clone` – The new clone.

void *`net_buf_simple_add`(struct *net_buf_simple* *buf, size_t len)

Prepare data to be added at the end of the buffer.

Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `len` – Number of bytes to increment the length with.

Returns The original tail of the buffer.

void *`net_buf_simple_add_mem`(struct *net_buf_simple* *buf, const void *mem, size_t len)

Copy given number of bytes from memory to the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `mem` – Location of data to be added.

- `len` – Length of data to be added

Returns The original tail of the buffer.

uint8_t *`net_buf_simple_add_u8`(struct *net_buf_simple* *buf, uint8_t val)

Add (8-bit) byte at the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – byte value to be added.

Returns Pointer to the value added

void `net_buf_simple_add_le16`(struct *net_buf_simple* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be added.

void `net_buf_simple_add_be16`(struct *net_buf_simple* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be added.

void `net_buf_simple_add_le24`(struct *net_buf_simple* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be added.

void `net_buf_simple_add_be24`(struct *net_buf_simple* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be added.

void `net_buf_simple_add_le32`(struct *net_buf_simple* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be added.

void `net_buf_simple_add_be32`(struct *net_buf_simple* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be added.

void `net_buf_simple_add_le48`(struct *net_buf_simple* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be added.

void `net_buf_simple_add_be48`(struct *net_buf_simple* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be added.

void `net_buf_simple_add_le64`(struct *net_buf_simple* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be added.

void `net_buf_simple_add_be64`(struct *net_buf_simple* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be added.

void *`net_buf_simple_remove_mem`(struct *net_buf_simple* *buf, size_t len)

Remove data from the end of the buffer.

Removes data from the end of the buffer by modifying the buffer length.

Parameters

- `buf` – Buffer to update.

- `len` – Number of bytes to remove.

Returns New end of the buffer data.

uint8_t `net_buf_simple_remove_u8`(struct *net_buf_simple* *buf)

Remove a 8-bit value from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 8-bit values.

Parameters

- `buf` – A valid pointer on a buffer.

Returns The 8-bit removed value

uint16_t `net_buf_simple_remove_le16`(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 16-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 16-bit value converted from little endian to host endian.

uint16_t `net_buf_simple_remove_be16`(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 16-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 16-bit value converted from big endian to host endian.

uint32_t `net_buf_simple_remove_le24`(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 24-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 24-bit value converted from little endian to host endian.

uint32_t `net_buf_simple_remove_be24`(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 24-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 24-bit value converted from big endian to host endian.

uint32_t `net_buf_simple_remove_le32`(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 32-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 32-bit value converted from little endian to host endian.

uint32_t `net_buf_simple_remove_be32`(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 32-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 32-bit value converted from big endian to host endian.

uint64_t `net_buf_simple_remove_le48`(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 48-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 48-bit value converted from little endian to host endian.

uint64_t `net_buf_simple_remove_be48`(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 48-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 48-bit value converted from big endian to host endian.

uint64_t `net_buf_simple_remove_le64`(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 64-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 64-bit value converted from little endian to host endian.

uint64_t `net_buf_simple_remove_be64`(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_simple_remove_mem()*, but a helper for operating on 64-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 64-bit value converted from big endian to host endian.

void *`net_buf_simple_push`(struct *net_buf_simple* *buf, size_t len)

Prepare data to be added to the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- `buf` – Buffer to update.

- `len` – Number of bytes to add to the beginning.

Returns The new beginning of the buffer data.

void *`net_buf_simple_push_mem`(struct *net_buf_simple* *buf, const void *mem, size_t len)

Copy given number of bytes from memory to the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `mem` – Location of data to be added.
- `len` – Length of data to be added.

Returns The new beginning of the buffer data.

void `net_buf_simple_push_le16`(struct *net_buf_simple* *buf, uint16_t val)

Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

void `net_buf_simple_push_be16`(struct *net_buf_simple* *buf, uint16_t val)

Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

void `net_buf_simple_push_u8`(struct *net_buf_simple* *buf, uint8_t val)

Push 8-bit value to the beginning of the buffer.

Adds 8-bit value the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 8-bit value to be pushed to the buffer.

void `net_buf_simple_push_le24`(struct *net_buf_simple* *buf, uint32_t val)

Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be pushed to the buffer.

void `net_buf_simple_push_be24`(struct *net_buf_simple* *buf, uint32_t val)

Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.

- `val` – 24-bit value to be pushed to the buffer.

void `net_buf_simple_push_le32`(struct *net_buf_simple* *buf, uint32_t val)

Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

void `net_buf_simple_push_be32`(struct *net_buf_simple* *buf, uint32_t val)

Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

void `net_buf_simple_push_le48`(struct *net_buf_simple* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be pushed to the buffer.

void `net_buf_simple_push_be48`(struct *net_buf_simple* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 48-bit value to be pushed to the buffer.

void `net_buf_simple_push_le64`(struct *net_buf_simple* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be pushed to the buffer.

void `net_buf_simple_push_be64`(struct *net_buf_simple* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 64-bit value to be pushed to the buffer.

void *net_buf_simple_pull(struct *net_buf_simple* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- buf – Buffer to update.
- len – Number of bytes to remove.

Returns New beginning of the buffer data.

void *net_buf_simple_pull_mem(struct *net_buf_simple* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- buf – Buffer to update.
- len – Number of bytes to remove.

Returns Pointer to the old location of the buffer data.

uint8_t net_buf_simple_pull_u8(struct *net_buf_simple* *buf)

Remove a 8-bit value from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 8-bit values.

Parameters

- buf – A valid pointer on a buffer.

Returns The 8-bit removed value

uint16_t net_buf_simple_pull_le16(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 16-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 16-bit value converted from little endian to host endian.

uint16_t net_buf_simple_pull_be16(struct *net_buf_simple* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 16-bit big endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 16-bit value converted from big endian to host endian.

uint32_t net_buf_simple_pull_le24(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 24-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 24-bit value converted from little endian to host endian.

uint32_t `net_buf_simple_pull_be24`(struct *net_buf_simple* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 24-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 24-bit value converted from big endian to host endian.

uint32_t `net_buf_simple_pull_le32`(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 32-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 32-bit value converted from little endian to host endian.

uint32_t `net_buf_simple_pull_be32`(struct *net_buf_simple* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 32-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 32-bit value converted from big endian to host endian.

uint64_t `net_buf_simple_pull_le48`(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 48-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 48-bit value converted from little endian to host endian.

uint64_t `net_buf_simple_pull_be48`(struct *net_buf_simple* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 48-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 48-bit value converted from big endian to host endian.

uint64_t `net_buf_simple_pull_le64`(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 64-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 64-bit value converted from little endian to host endian.

uint64_t net_buf_simple_pull_be64(struct *net_buf_simple* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_simple_pull()*, but a helper for operating on 64-bit big endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 64-bit value converted from big endian to host endian.

static inline uint8_t *net_buf_simple_tail(struct *net_buf_simple* *buf)

Get the tail pointer for a buffer.

Get a pointer to the end of the data in a buffer.

Parameters

- buf – Buffer.

Returns Tail pointer for the buffer.

size_t net_buf_simple_headroom(struct *net_buf_simple* *buf)

Check buffer headroom.

Check how much free space there is in the beginning of the buffer.

buf A valid pointer on a buffer

Returns Number of bytes available in the beginning of the buffer.

size_t net_buf_simple_tailroom(struct *net_buf_simple* *buf)

Check buffer tailroom.

Check how much free space there is at the end of the buffer.

Parameters

- buf – A valid pointer on a buffer

Returns Number of bytes available at the end of the buffer.

uint16_t net_buf_simple_max_len(struct *net_buf_simple* *buf)

Check maximum *net_buf_simple::len* value.

This value is depending on the number of bytes being reserved as headroom.

Parameters

- buf – A valid pointer on a buffer

Returns Number of bytes usable behind the *net_buf_simple::data* pointer.

static inline void net_buf_simple_save(struct *net_buf_simple* *buf, struct *net_buf_simple_state* *state)

Save the parsing state of a buffer.

Saves the parsing state of a buffer so it can be restored later.

Parameters

- buf – Buffer from which the state should be saved.
- state – Storage for the state.

static inline void net_buf_simple_restore(struct *net_buf_simple* *buf, struct *net_buf_simple_state* *state)

Restore the parsing state of a buffer.

Restores the parsing state of a buffer from a state previously stored by *net_buf_simple_save()*.

Parameters

- `buf` – Buffer to which the state should be restored.
- `state` – Stored state.

struct *net_buf_pool* *`net_buf_pool_get`(int id)

Looks up a pool based on its ID.

Parameters

- `id` – Pool ID (e.g. from buf->pool_id).

Returns Pointer to pool.

int `net_buf_id`(struct *net_buf* *buf)

Get a zero-based index for a buffer.

This function will translate a buffer into a zero-based index, based on its placement in its buffer pool. This can be useful if you want to associate an external array of meta-data contexts with the buffers of a pool.

Parameters

- `buf` – Network buffer.

Returns Zero-based index for the buffer.

struct *net_buf* *`net_buf_alloc_fixed`(struct *net_buf_pool* *pool, *k_timeout_t* timeout)

Allocate a new fixed buffer from a pool.

Parameters

- `pool` – Which pool to allocate the buffer from.
- `timeout` – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout. Note that some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Returns New buffer or NULL if out of buffers.

static inline struct *net_buf* *`net_buf_alloc`(struct *net_buf_pool* *pool, *k_timeout_t* timeout)

Parameters

- `pool` – Which pool to allocate the buffer from.
- `timeout` – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout. Note that some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Returns New buffer or NULL if out of buffers.

struct *net_buf* *`net_buf_alloc_len`(struct *net_buf_pool* *pool, size_t size, *k_timeout_t* timeout)

Allocate a new variable length buffer from a pool.

Parameters

- `pool` – Which pool to allocate the buffer from.
- `size` – Amount of data the buffer must be able to fit.
- `timeout` – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout. Note that some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Returns New buffer or NULL if out of buffers.

struct *net_buf* *net_buf_alloc_with_data(struct *net_buf_pool* *pool, void *data, size_t size, *k_timeout_t* timeout)

Allocate a new buffer from a pool but with external data pointer.

Allocate a new buffer from a pool, where the data pointer comes from the user and not from the pool.

Parameters

- pool – Which pool to allocate the buffer from.
- data – External data pointer
- size – Amount of data the pointed data buffer if able to fit.
- timeout – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout. Note that some types of data allocators do not support blocking (such as the HEAP type). In this case it's still possible for *net_buf_alloc()* to fail (return NULL) even if it was given K_FOREVER.

Returns New buffer or NULL if out of buffers.

struct *net_buf* *net_buf_get(struct k_fifo *fifo, *k_timeout_t* timeout)

Get a buffer from a FIFO.

This function is NOT thread-safe if the buffers in the FIFO contain fragments.

Parameters

- fifo – Which FIFO to take the buffer from.
- timeout – Affects the action taken should the FIFO be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns New buffer or NULL if the FIFO is empty.

static inline void net_buf_destroy(struct *net_buf* *buf)

Destroy buffer from custom destroy callback.

This helper is only intended to be used from custom destroy callbacks. If no custom destroy callback is given to NET_BUF_POOL_*_DEFINE() then there is no need to use this API.

Parameters

- buf – Buffer to destroy.

void net_buf_reset(struct *net_buf* *buf)

Reset buffer.

Reset buffer data and flags so it can be reused for other purposes.

Parameters

- buf – Buffer to reset.

void net_buf_simple_reserve(struct *net_buf_simple* *buf, size_t reserve)

Initialize buffer with the given headroom.

The buffer is not expected to contain any data when this API is called.

Parameters

- buf – Buffer to initialize.
- reserve – How much headroom to reserve.

void net_buf_slist_put(sys_slist_t *list, struct *net_buf* *buf)

Put a buffer into a list.

If the buffer contains follow-up fragments this function will take care of inserting them as well into the list.

Parameters

- list – Which list to append the buffer to.
- buf – Buffer.

struct *net_buf* *net_buf_slist_get(sys_slist_t *list)

Get a buffer from a list.

If the buffer had any fragments, these will automatically be recovered from the list as well and be placed to the buffer's fragment list. This function is NOT thread-safe when recovering fragments.

Parameters

- list – Which list to take the buffer from.

Returns New buffer or NULL if the FIFO is empty.

void net_buf_put(struct k_fifo *fifo, struct *net_buf* *buf)

Put a buffer to the end of a FIFO.

If the buffer contains follow-up fragments this function will take care of inserting them as well into the FIFO.

Parameters

- fifo – Which FIFO to put the buffer to.
- buf – Buffer.

void net_buf_unref(struct *net_buf* *buf)

Decrements the reference count of a buffer.

The buffer is put back into the pool if the reference count reaches zero.

Parameters

- buf – A valid pointer on a buffer

struct *net_buf* *net_buf_ref(struct *net_buf* *buf)

Increment the reference count of a buffer.

Parameters

- buf – A valid pointer on a buffer

Returns the buffer newly referenced

struct *net_buf* *net_buf_clone(struct *net_buf* *buf, *k_timeout_t* timeout)

Clone buffer.

Duplicate given buffer including any data and headers currently stored.

Parameters

- buf – A valid pointer on a buffer
- timeout – Affects the action taken should the pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait until the specified timeout.

Returns Cloned buffer or NULL if out of buffers.

static inline void *net_buf_user_data(const struct *net_buf* *buf)

Get a pointer to the user data of a buffer.

Parameters

- buf – A valid pointer on a buffer

Returns Pointer to the user data of the buffer.

static inline void net_buf_reserve(struct *net_buf* *buf, size_t reserve)

Initialize buffer with the given headroom.

The buffer is not expected to contain any data when this API is called.

Parameters

- buf – Buffer to initialize.
- reserve – How much headroom to reserve.

static inline void *net_buf_add(struct *net_buf* *buf, size_t len)

Prepare data to be added at the end of the buffer.

Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- len – Number of bytes to increment the length with.

Returns The original tail of the buffer.

static inline void *net_buf_add_mem(struct *net_buf* *buf, const void *mem, size_t len)

Copies the given number of bytes to the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- mem – Location of data to be added.
- len – Length of data to be added

Returns The original tail of the buffer.

static inline uint8_t *net_buf_add_u8(struct *net_buf* *buf, uint8_t val)

Add (8-bit) byte at the end of the buffer.

Increments the data length of the buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – byte value to be added.

Returns Pointer to the value added

static inline void net_buf_add_le16(struct *net_buf* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 16-bit value to be added.

static inline void net_buf_add_be16(struct *net_buf* *buf, uint16_t val)

Add 16-bit value at the end of the buffer.

Adds 16-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 16-bit value to be added.

static inline void net_buf_add_le24(struct *net_buf* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 24-bit value to be added.

static inline void net_buf_add_be24(struct *net_buf* *buf, uint32_t val)

Add 24-bit value at the end of the buffer.

Adds 24-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 24-bit value to be added.

static inline void net_buf_add_le32(struct *net_buf* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 32-bit value to be added.

static inline void net_buf_add_be32(struct *net_buf* *buf, uint32_t val)

Add 32-bit value at the end of the buffer.

Adds 32-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 32-bit value to be added.

static inline void net_buf_add_le48(struct *net_buf* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 48-bit value to be added.

static inline void net_buf_add_be48(struct *net_buf* *buf, uint64_t val)

Add 48-bit value at the end of the buffer.

Adds 48-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 48-bit value to be added.

static inline void net_buf_add_le64(struct *net_buf* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in little endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 64-bit value to be added.

static inline void net_buf_add_be64(struct *net_buf* *buf, uint64_t val)

Add 64-bit value at the end of the buffer.

Adds 64-bit value in big endian format at the end of buffer. Increments the data length of a buffer to account for more data at the end.

Parameters

- buf – Buffer to update.
- val – 64-bit value to be added.

static inline void *net_buf_remove_mem(struct *net_buf* *buf, size_t len)

Remove data from the end of the buffer.

Removes data from the end of the buffer by modifying the buffer length.

Parameters

- buf – Buffer to update.
- len – Number of bytes to remove.

Returns New end of the buffer data.

static inline uint8_t net_buf_remove_u8(struct *net_buf* *buf)

Remove a 8-bit value from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 8-bit values.

Parameters

- buf – A valid pointer on a buffer.

Returns The 8-bit removed value

static inline uint16_t net_buf_remove_le16(struct *net_buf* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 16-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 16-bit value converted from little endian to host endian.

static inline uint16_t net_buf_remove_be16(struct *net_buf* *buf)

Remove and convert 16 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 16-bit big endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 16-bit value converted from big endian to host endian.

static inline uint32_t net_buf_remove_be24(struct *net_buf* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 24-bit big endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 24-bit value converted from big endian to host endian.

static inline uint32_t net_buf_remove_le24(struct *net_buf* *buf)

Remove and convert 24 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 24-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 24-bit value converted from little endian to host endian.

static inline uint32_t net_buf_remove_le32(struct *net_buf* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 32-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 32-bit value converted from little endian to host endian.

static inline uint32_t net_buf_remove_be32(struct *net_buf* *buf)

Remove and convert 32 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 32-bit big endian data.

Parameters

- buf – A valid pointer on a buffer

Returns 32-bit value converted from big endian to host endian.

static inline uint64_t net_buf_remove_le48(struct *net_buf* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 48-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 48-bit value converted from little endian to host endian.

static inline uint64_t net_buf_remove_be48(struct *net_buf* *buf)

Remove and convert 48 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 48-bit big endian data.

Parameters

- buf – A valid pointer on a buffer

Returns 48-bit value converted from big endian to host endian.

static inline uint64_t net_buf_remove_le64(struct *net_buf* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 64-bit little endian data.

Parameters

- buf – A valid pointer on a buffer.

Returns 64-bit value converted from little endian to host endian.

static inline uint64_t net_buf_remove_be64(struct *net_buf* *buf)

Remove and convert 64 bits from the end of the buffer.

Same idea as with *net_buf_remove_mem()*, but a helper for operating on 64-bit big endian data.

Parameters

- buf – A valid pointer on a buffer

Returns 64-bit value converted from big endian to host endian.

static inline void *net_buf_push(struct *net_buf* *buf, size_t len)

Prepare data to be added at the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- buf – Buffer to update.
- len – Number of bytes to add to the beginning.

Returns The new beginning of the buffer data.

static inline void *net_buf_push_mem(struct *net_buf* *buf, const void *mem, size_t len)

Copies the given number of bytes to the start of the buffer.

Modifies the data pointer and buffer length to account for more data in the beginning of the buffer.

Parameters

- buf – Buffer to update.
- mem – Location of data to be added.
- len – Length of data to be added.

Returns The new beginning of the buffer data.

static inline void net_buf_push_u8(struct *net_buf* *buf, uint8_t val)

Push 8-bit value to the beginning of the buffer.

Adds 8-bit value the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 8-bit value to be pushed to the buffer.

static inline void `net_buf_push_le16`(struct *net_buf* *buf, uint16_t val)

Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

static inline void `net_buf_push_be16`(struct *net_buf* *buf, uint16_t val)

Push 16-bit value to the beginning of the buffer.

Adds 16-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 16-bit value to be pushed to the buffer.

static inline void `net_buf_push_le24`(struct *net_buf* *buf, uint32_t val)

Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be pushed to the buffer.

static inline void `net_buf_push_be24`(struct *net_buf* *buf, uint32_t val)

Push 24-bit value to the beginning of the buffer.

Adds 24-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 24-bit value to be pushed to the buffer.

static inline void `net_buf_push_le32`(struct *net_buf* *buf, uint32_t val)

Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in little endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

static inline void `net_buf_push_be32`(struct *net_buf* *buf, uint32_t val)

Push 32-bit value to the beginning of the buffer.

Adds 32-bit value in big endian format to the beginning of the buffer.

Parameters

- `buf` – Buffer to update.
- `val` – 32-bit value to be pushed to the buffer.

static inline void net_buf_push_le48(struct *net_buf* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in little endian format to the beginning of the buffer.

Parameters

- buf – Buffer to update.
- val – 48-bit value to be pushed to the buffer.

static inline void net_buf_push_be48(struct *net_buf* *buf, uint64_t val)

Push 48-bit value to the beginning of the buffer.

Adds 48-bit value in big endian format to the beginning of the buffer.

Parameters

- buf – Buffer to update.
- val – 48-bit value to be pushed to the buffer.

static inline void net_buf_push_le64(struct *net_buf* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in little endian format to the beginning of the buffer.

Parameters

- buf – Buffer to update.
- val – 64-bit value to be pushed to the buffer.

static inline void net_buf_push_be64(struct *net_buf* *buf, uint64_t val)

Push 64-bit value to the beginning of the buffer.

Adds 64-bit value in big endian format to the beginning of the buffer.

Parameters

- buf – Buffer to update.
- val – 64-bit value to be pushed to the buffer.

static inline void *net_buf_pull(struct *net_buf* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- buf – Buffer to update.
- len – Number of bytes to remove.

Returns New beginning of the buffer data.

static inline void *net_buf_pull_mem(struct *net_buf* *buf, size_t len)

Remove data from the beginning of the buffer.

Removes data from the beginning of the buffer by modifying the data pointer and buffer length.

Parameters

- buf – Buffer to update.
- len – Number of bytes to remove.

Returns Pointer to the old beginning of the buffer data.

static inline uint8_t `net_buf_pull_u8`(struct *net_buf* *buf)

Remove a 8-bit value from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 8-bit values.

Parameters

- `buf` – A valid pointer on a buffer.

Returns The 8-bit removed value

static inline uint16_t `net_buf_pull_le16`(struct *net_buf* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 16-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 16-bit value converted from little endian to host endian.

static inline uint16_t `net_buf_pull_be16`(struct *net_buf* *buf)

Remove and convert 16 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 16-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 16-bit value converted from big endian to host endian.

static inline uint32_t `net_buf_pull_le24`(struct *net_buf* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 24-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 24-bit value converted from little endian to host endian.

static inline uint32_t `net_buf_pull_be24`(struct *net_buf* *buf)

Remove and convert 24 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 24-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 24-bit value converted from big endian to host endian.

static inline uint32_t `net_buf_pull_le32`(struct *net_buf* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 32-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 32-bit value converted from little endian to host endian.

static inline uint32_t `net_buf_pull_be32`(struct *net_buf* *buf)

Remove and convert 32 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 32-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer

Returns 32-bit value converted from big endian to host endian.

static inline uint64_t `net_buf_pull_le48`(struct *net_buf* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 48-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 48-bit value converted from little endian to host endian.

static inline uint64_t `net_buf_pull_be48`(struct *net_buf* *buf)

Remove and convert 48 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 48-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer

Returns 48-bit value converted from big endian to host endian.

static inline uint64_t `net_buf_pull_le64`(struct *net_buf* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 64-bit little endian data.

Parameters

- `buf` – A valid pointer on a buffer.

Returns 64-bit value converted from little endian to host endian.

static inline uint64_t `net_buf_pull_be64`(struct *net_buf* *buf)

Remove and convert 64 bits from the beginning of the buffer.

Same idea as with *net_buf_pull()*, but a helper for operating on 64-bit big endian data.

Parameters

- `buf` – A valid pointer on a buffer

Returns 64-bit value converted from big endian to host endian.

static inline size_t `net_buf_tailroom`(struct *net_buf* *buf)

Check buffer tailroom.

Check how much free space there is at the end of the buffer.

Parameters

- `buf` – A valid pointer on a buffer

Returns Number of bytes available at the end of the buffer.

static inline size_t `net_buf_headroom`(struct *net_buf* *buf)

Check buffer headroom.

Check how much free space there is in the beginning of the buffer.

buf A valid pointer on a buffer

Returns Number of bytes available in the beginning of the buffer.

static inline uint16_t net_buf_max_len(struct *net_buf* *buf)

Check maximum *net_buf::len* value.

This value is depending on the number of bytes being reserved as headroom.

Parameters

- buf – A valid pointer on a buffer

Returns Number of bytes usable behind the *net_buf::data* pointer.

static inline uint8_t *net_buf_tail(struct *net_buf* *buf)

Get the tail pointer for a buffer.

Get a pointer to the end of the data in a buffer.

Parameters

- buf – Buffer.

Returns Tail pointer for the buffer.

struct *net_buf* *net_buf_frag_last(struct *net_buf* *frags)

Find the last fragment in the fragment list.

Returns Pointer to last fragment in the list.

void net_buf_frag_insert(struct *net_buf* *parent, struct *net_buf* *frag)

Insert a new fragment to a chain of bufs.

Insert a new fragment into the buffer fragments list after the parent.

Note: This function takes ownership of the fragment reference so the caller is not required to unref.

Parameters

- parent – Parent buffer/fragment.
- frag – Fragment to insert.

struct *net_buf* *net_buf_frag_add(struct *net_buf* *head, struct *net_buf* *frag)

Add a new fragment to the end of a chain of bufs.

Append a new fragment into the buffer fragments list.

Note: This function takes ownership of the fragment reference so the caller is not required to unref.

Parameters

- head – Head of the fragment chain.
- frag – Fragment to add.

Returns New head of the fragment chain. Either head (if head was non-NULL) or frag (if head was NULL).

struct *net_buf* *net_buf_frag_del(struct *net_buf* *parent, struct *net_buf* *frag)

Delete existing fragment from a chain of bufs.

Parameters

- parent – Parent buffer/fragment, or NULL if there is no parent.
- frag – Fragment to delete.

Returns Pointer to the buffer following the fragment, or NULL if it had no further fragments.

size_t net_buf_linearize(void *dst, size_t dst_len, struct *net_buf* *src, size_t offset, size_t len)

Copy bytes from *net_buf* chain starting at offset to linear buffer.

Copy (extract) *len* bytes from *src* *net_buf* chain, starting from *offset* in it, to a linear buffer *dst*. Return number of bytes actually copied, which may be less than requested, if *net_buf* chain doesn't have enough data, or destination buffer is too small.

Parameters

- `dst` – Destination buffer
- `dst_len` – Destination buffer length
- `src` – Source *net_buf* chain
- `offset` – Starting offset to copy from
- `len` – Number of bytes to copy

Returns number of bytes actually copied

size_t net_buf_append_bytes(struct *net_buf* *buf, size_t len, const void *value, *k_timeout_t* timeout, *net_buf_allocator_cb* allocate_cb, void *user_data)

Append data to a list of *net_buf*.

Append data to a *net_buf*. If there is not enough space in the *net_buf* then more *net_buf* will be added, unless there are no free *net_buf* and timeout occurs. If not allocator is provided it attempts to allocate from the same pool as the original buffer.

Parameters

- `buf` – Network buffer.
- `len` – Total length of input data
- `value` – Data to be added
- `timeout` – Timeout is passed to the *net_buf* allocator callback.
- `allocate_cb` – When a new *net_buf* is required, use this callback.
- `user_data` – A user data pointer to be supplied to the allocate_cb. This pointer is can be anything from a mem_pool or a *net_pkt*, the logic is left up to the allocate_cb function.

Returns Length of data actually added. This may be less than input length if other timeout than K_FOREVER was used, and there were no free fragments in a pool to accommodate all data.

static inline struct *net_buf* *net_buf_skip(struct *net_buf* *buf, size_t len)

Skip N number of bytes in a *net_buf*.

Skip N number of bytes starting from fragment's offset. If the total length of data is placed in multiple fragments, this function will skip from all fragments until it reaches N number of bytes. Any fully skipped buffers are removed from the *net_buf* list.

Parameters

- `buf` – Network buffer.
- `len` – Total length of data to be skipped.

Returns Pointer to the fragment or NULL and pos is 0 after successful skip, NULL and pos is 0xffff otherwise.

static inline size_t net_buf_frags_len(struct *net_buf* *buf)

Calculate amount of bytes stored in fragments.

Calculates the total amount of data stored in the given buffer and the fragments linked to it.

Parameters

- `buf` – Buffer to start off with.

Returns Number of bytes in the buffer and its fragments.

struct `net_buf_simple`

#include <buf.h> Simple network buffer representation.

This is a simpler variant of the *net_buf* object (in fact *net_buf* uses *net_buf_simple* internally). It doesn't provide any kind of reference counting, user data, dynamic allocation, or in general the ability to pass through kernel objects such as FIFOs.

The main use of this is for scenarios where the meta-data of the normal *net_buf* isn't needed and causes too much overhead. This could be e.g. when the buffer only needs to be allocated on the stack or when the access to and lifetime of the buffer is well controlled and constrained.

Public Members

uint8_t *`data`

Pointer to the start of data in the buffer.

uint16_t `len`

Length of the data behind the data pointer.

To determine the max length, use *net_buf_simple_max_len()*, not *size*!

uint16_t `size`

Amount of data that net_buf_simple::__buf can store.

struct `net_buf_simple_state`

#include <buf.h> Parsing state of a buffer.

This is used for temporarily storing the parsing state of a buffer while giving control of the parsing to a routine which we don't control.

Public Members

uint16_t `offset`

Offset of the data pointer from the beginning of the storage

uint16_t `len`

Length of data

struct `net_buf`

#include <buf.h> Network buffer representation.

This struct is used to represent network buffers. Such buffers are normally defined through the NET_BUF_POOL_*_DEFINE() APIs and allocated using the *net_buf_alloc()* API.

Public Members

sys_snode_t `node`

Allow placing the buffer into sys_slist_t

struct *net_buf* *frags
: Fragments associated with this buffer.

uint8_t ref
: Reference count.

uint8_t flags
: Bit-field of buffer flags.

uint8_t pool_id
: Where the buffer should go when freed up.

uint8_t *data
: Pointer to the start of data in the buffer.

uint16_t len
: Length of the data behind the data pointer.

uint16_t size
: Amount of data that this buffer can store.

uint8_t user_data[0]
: System metadata for this buffer.

struct net_buf_data_cb
: *#include <buf.h>*

struct net_buf_data_alloc
: *#include <buf.h>*

struct net_buf_pool
: *#include <buf.h>* Network buffer pool representation.

This struct is used to represent a pool of network buffers.

Public Members

struct k_lifo free
: LIFO to place the buffer into when free

const uint16_t buf_count
: Number of buffers in pool

uint16_t uninit_count
: Number of uninitialized buffers

void (*const destroy)(struct *net_buf* *buf)
: Optional destroy callback when buffer is freed.

const struct *net_buf_data_alloc* *alloc

Data allocation handlers.

struct net_buf_pool_fixed

#include <buf.h>

Packet Management

Overview Network packets are the main data the networking stack manipulates. Such data is represented through the net_pkt structure which provides a means to hold the packet, write and read it, as well as necessary metadata for the core to hold important information. Such an object is called net_pkt in this document.

The data structure and the whole API around it are defined in include/net/net_pkt.h.

Architectural notes There are two network packets flows within the stack, **TX** for the transmission path, and **RX** for the reception one. In both paths, each net_pkt is written and read from the beginning to the end, or more specifically from the headers to the payload.

Memory management

Allocation All net_pkt objects come from a pre-defined pool of struct net_pkt. Such pool is defined via

```
NET_PKT_SLAB_DEFINE(name, count)
```

Note, however, one will rarely have to use it, as the core provides already two pools, one for the TX path and one for the RX path.

Allocating a raw net_pkt can be done through:

```
pkt = net_pkt_alloc(timeout);
```

However, by its nature, a raw net_pkt is useless without a buffer and needs various metadata information to become relevant as well. It requires at least to get the network interface it is meant to be sent through or through which it was received. As this is a very common operation, a helper exist:

```
pkt = net_pkt_alloc_on_iface(iface, timeout);
```

A more complete allocator exists, where both the net_pkt and its buffer can be allocated at once:

```
pkt = net_pkt_alloc_with_buffer(iface, size, family, proto, timeout);
```

See below how the buffer is allocated.

Buffer allocation The net_pkt object does not define its own buffer, but instead uses an existing object for this: *net_buf*. (See *Network Buffer* for more information). However, it mostly hides the usage of such a buffer because net_pkt brings network awareness to buffer allocation and, as we will see later, its operation too.

To allocate a buffer, a net_pkt needs to have at least its network interface set. This works if the family of the packet is unknown at the time of buffer allocation. Then one could do:

```
net_pkt_alloc_buffer(pkt, size, proto, timeout);
```

Where proto could be 0 if unknown (there is no IPPROTO_UNSPEC).

As seen previously, the net_pkt and its buffer can be allocated at once via *net_pkt_alloc_with_buffer()*. It is actually the most widely used allocator.

The network interface, the family, and the protocol of the packet are used by the buffer allocation to determine if the requested size can be allocated. Indeed, the allocator will use the network interface to know the MTU and then the family and protocol for the headers space (if only these 2 are specified). If the whole fits within the MTU, the allocated space will be of the requested size plus, eventually, the headers space. If there is insufficient MTU space, the requested size will be shrunk so the possible headers space and new size will fit within the MTU.

For instance, on an Ethernet network interface, with an MTU of 1500 bytes:

```
pkt = net_pkt_alloc_with_buffer(iface, 800, AF_INET4, IPPROTO_UDP, K_FOREVER);
```

will successfully allocate 800 + 20 + 8 bytes of buffer for the new net_pkt where:

```
pkt = net_pkt_alloc_with_buffer(iface, 1600, AF_INET4, IPPROTO_UDP, K_FOREVER);
```

will successfully allocate 1500 bytes, and where 20 + 8 bytes (IPv4 + UDP headers) will not be used for the payload.

On the receiving side, when the family and protocol are not known:

```
pkt = net_pkt_rx_alloc_with_buffer(iface, 800, AF_UNSPEC, 0, K_FOREVER);
```

will allocate 800 bytes and no extra header space. But a:

```
pkt = net_pkt_rx_alloc_with_buffer(iface, 1600, AF_UNSPEC, 0, K_FOREVER);
```

will allocate 1514 bytes, the MTU + Ethernet header space.

One can increase the amount of buffer space allocated by calling *net_pkt_alloc_buffer()*, as it will take into account the existing buffer. It will also account for the header space if net_pkt's family is a valid one, as well as the proto parameter. In that case, the newly allocated buffer space will be appended to the existing one, and not inserted in the front. Note however such a use case is rather limited. Usually, one should know from the start how much size should be requested.

Deallocation Each net_pkt is reference counted. At allocation, the reference is set to 1. The reference count can be incremented with *net_pkt_ref()* or decremented with *net_pkt_unref()*. When the count drops to zero the buffer is also un-referenced and net_pkt is automatically placed back into the free net_pkt_slabs

If net_pkt's buffer is needed even after net_pkt deallocation, one will need to reference once more all the chain of net_buf before calling last net_pkt_unref. See *Network Buffer* for more information.

Operations There are two ways to access the net_pkt buffer, explained in the following sections: basic read/write access and data access, the latter being the preferred way.

Read and Write access As said earlier, though net_pkt uses net_buf for its buffer, it provides its own API to access it. Indeed, a network packet might be scattered over a chain of net_buf objects, the functions provided by net_buf are then limited for such case. Instead, net_pkt provides functions which hide all the complexity of potential non-contiguous access.

Data movement into the buffer is made through a cursor maintained within each net_pkt. All read/write operations affect this cursor. Note as well that read or write functions are strict on their length parameters: if it cannot r/w the given length it will fail. Length is not interpreted as an upper limit, it is instead the exact amount of data that must be read or written.

As there are two paths, TX and RX, there are two access modes: write and overwrite. This might sound a bit unusual, but is in fact simple and provides flexibility.

In write mode, whatever is written in the buffer affects the length of actual data present in the buffer. Buffer length should not be confused with the buffer size which is a limit any mode cannot pass. In overwrite mode then, whatever is written must happen on valid data, and will not affect the buffer length. By default, a newly allocated net_pkt is on write mode, and its cursor points to the beginning of its buffer.

Let's see now, step by step, the functions and how they behave depending on the mode.

When freshly allocated with a buffer of 500 bytes, a net_pkt has 0 length, which means no valid data is in its buffer. One could verify this by:

```
len = net_pkt_get_len(pkt);
```

Now, let's write 8 bytes:

```
net_pkt_write(pkt, data, 8);
```

The buffer length is now 8 bytes. There are various helpers to write a byte, or big endian uint16_t, uint32_t.

```
net_pkt_write_u8(pkt, &foo);
net_pkt_write_be16(pkt, &ba);
net_pkt_write_be32(pkt, &bar);
```

Logically, net_pkt's length is now 15. But if we try to read at this point, it will fail because there is nothing to read at the cursor where we are at in the net_pkt. It is possible, while in write mode, to read what has been already written by resetting the cursor of the net_pkt. For instance:

```
net_pkt_cursor_init(pkt);
net_pkt_read(pkt, data, 15);
```

This will reset the cursor of the pkt to the beginning of the buffer and then let you read the actual 15 bytes present. The cursor is then again pointing at the end of the buffer.

To set a large area with the same byte, a memset function is provided:

```
net_pkt_memset(pkt, 0, 5);
```

Our net_pkt has now a length of 20 bytes.

Switching between modes can be achieved via `net_pkt_set_overwrite()` function. It is possible to switch mode back and forth at any time. The net_pkt will be set to overwrite and its cursor reset:

```
net_pkt_set_overwrite(pkt, true);
net_pkt_cursor_init(pkt);
```

Now the same operators can be used, but it will be limited to the existing data in the buffer, i.e. 20 bytes.

If it is necessary to know how much space is available in the net_pkt call:

```
net_pkt_available_buffer(pkt);
```

Or, if headers space needs to be accounted for, call:

```
net_pkt_available_payload_buffer(pkt, proto);
```

If you want to place the cursor at a known position use the function *net_pkt_skip()*. For example, to go after the IP header, use:

```
net_pkt_cursor_init(pkt);
net_pkt_skip(pkt, net_pkt_ip_header_len(pkt));
```

Data access Though the API shown previously is rather simple, it involves always copying things to and from the net_pkt buffer. In many occasions, it is more relevant to access the information stored in the buffer contiguously, especially with network packets which embed headers.

These headers are, most of the time, a known fixed set of bytes. It is then more natural to have a structure representing a certain type of header. In addition to this, if it is known the header size appears in a contiguous area of the buffer, it will be way more efficient to cast the actual position in the buffer to the type of header. Either for reading or writing the fields of such header, accessing it directly will save memory.

Net pkt comes with a dedicated API for this, built on top of the previously described API. It is able to handle both contiguous and non-contiguous access transparently.

There are two macros used to define a data access descriptor: *NET_PKT_DATA_ACCESS_DEFINE* when it is not possible to tell if the data will be in a contiguous area, and *NET_PKT_DATA_ACCESS_CONTIGUOUS_DEFINE* when it is guaranteed the data is in a contiguous area.

Let's take the example of IP and UDP. Both IPv4 and IPv6 headers are always found at the beginning of the packet and are small enough to fit in a net_buf of 128 bytes (for instance, though 64 bytes could be chosen).

```
NET_PKT_DATA_ACCESS_CONTIGUOUS_DEFINE(ipv4_access, struct net_ipv4_hdr);
struct net_ipv4_hdr *ipv4_hdr;

ipv4_hdr = (struct net_ipv4_hdr *)net_pkt_get_data(pkt, &ipv4_acess);
```

It would be the same for struct net_ipv4_hdr. For a UDP header it is likely not to be in a contiguous area in IPv6 for instance so:

```
NET_PKT_DATA_ACCESS_DEFINE(udp_access, struct net_udp_hdr);
struct net_udp_hdr *udp_hdr;

udp_hdr = (struct net_udp_hdr *)net_pkt_get_data(pkt, &udp_access);
```

At this point, the cursor of the net_pkt points at the beginning of the requested data. On the RX path, these headers will be read but not modified so to proceed further the cursor needs to advance past the data. There is a function dedicated for this:

```
net_pkt_acknowledge_data(pkt, &ipv4_access);
```

On the TX path, however, the header fields have been modified. In such a case:

```
net_pkt_set_data(pkt, &ipv4_access);
```

If the data are in a contiguous area, it will advance the cursor relevantly. If not, it will write the data and the cursor will be updated. Note that *net_pkt_set_data()* could be used in the RX path as well, but it is slightly faster to use *net_pkt_acknowledge_data()* as this one does not care about contiguity at all, it just advances the cursor via *net_pkt_skip()* directly.

API Reference

group net_pkt

Network packet management library.

Defines

NET_PKT_SLAB_DEFINE(name, count)

Create a *net_pkt* slab.

A *net_pkt* slab is used to store meta-information about network packets. It must be coupled with a data fragment pool (:c:macro:NET_PKT_DATA_POOL_DEFINE) used to store the actual packet data. The macro can be used by an application to define additional custom per-context TX packet slabs (see :c:func:net_context_setup_pools).

Parameters

- name – Name of the slab.
- count – Number of *net_pkt* in this slab.

NET_PKT_TX_SLAB_DEFINE(name, count)

NET_PKT_DATA_POOL_DEFINE(name, count)

Create a data fragment *net_buf* pool.

A *net_buf* pool is used to store actual data for network packets. It must be coupled with a *net_pkt* slab (:c:macro:NET_PKT_SLAB_DEFINE) used to store the packet meta-information. The macro can be used by an application to define additional custom per-context TX packet pools (see :c:func:net_context_setup_pools).

Parameters

- name – Name of the pool.
- count – Number of *net_buf* in this pool.

net_pkt_print_frags(pkt)

Print fragment list and the fragment sizes.

Only available if debugging is activated.

Parameters

- pkt – Network pkt.

NET_PKT_DATA_ACCESS_DEFINE(_name, _type)

NET_PKT_DATA_ACCESS_CONTIGUOUS_DEFINE(_name, _type)

Functions

struct *net_buf* *net_pkt_get_reserve_rx_data(*k_timeout_t* timeout)

Get RX DATA buffer from pool. Normally you should use *net_pkt_get_frag()* instead.

Normally this version is not useful for applications but is mainly used by network fragmentation code.

Parameters

- timeout – Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait up to the specified time.

Returns Network buffer if successful, NULL otherwise.

struct *net_buf* *net_pkt_get_reserve_tx_data(*k_timeout_t* timeout)

Get TX DATA buffer from pool. Normally you should use *net_pkt_get_frag()* instead.

Normally this version is not useful for applications but is mainly used by network fragmentation code.

Parameters

- timeout – Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait up to the specified time.

Returns Network buffer if successful, NULL otherwise.

struct *net_buf* *net_pkt_get_frag(struct *net_pkt* *pkt, *k_timeout_t* timeout)

Get a data fragment that might be from user specific buffer pool or from global DATA pool.

Parameters

- pkt – Network packet.
- timeout – Affects the action taken should the net buf pool be empty. If K_NO_WAIT, then return immediately. If K_FOREVER, then wait as long as necessary. Otherwise, wait up to the specified time.

Returns Network buffer if successful, NULL otherwise.

void net_pkt_unref(struct *net_pkt* *pkt)

Place packet back into the available packets slab.

Releases the packet to other use. This needs to be called by application after it has finished with the packet.

Parameters

- pkt – Network packet to release.

struct *net_pkt* *net_pkt_ref(struct *net_pkt* *pkt)

Increase the packet ref count.

Mark the packet to be used still.

Parameters

- pkt – Network packet to ref.

Returns Network packet if successful, NULL otherwise.

struct *net_buf* *net_pkt_frag_ref(struct *net_buf* *frag)

Increase the packet fragment ref count.

Mark the fragment to be used still.

Parameters

- frag – Network fragment to ref.

Returns a pointer on the referenced Network fragment.

void `net_pkt_frag_unref`(struct *net_buf* *frag)

Decrease the packet fragment ref count.

Parameters

- `frag` – Network fragment to unref.

struct *net_buf* *`net_pkt_frag_del`(struct *net_pkt* *pkt, struct *net_buf* *parent, struct *net_buf* *frag)

Delete existing fragment from a packet.

Parameters

- `pkt` – Network packet from which frag belongs to.
- `parent` – parent fragment of frag, or NULL if none.
- `frag` – Fragment to delete.

Returns Pointer to the following fragment, or NULL if it had no further fragments.

void `net_pkt_frag_add`(struct *net_pkt* *pkt, struct *net_buf* *frag)

Add a fragment to a packet at the end of its fragment list.

Parameters

- `pkt` – pkt Network packet where to add the fragment
- `frag` – Fragment to add

void `net_pkt_frag_insert`(struct *net_pkt* *pkt, struct *net_buf* *frag)

Insert a fragment to a packet at the beginning of its fragment list.

Parameters

- `pkt` – pkt Network packet where to insert the fragment
- `frag` – Fragment to insert

bool `net_pkt_compact`(struct *net_pkt* *pkt)

Compact the fragment list of a packet.

After this there is no more any free space in individual fragments.

Parameters

- `pkt` – Network packet.

Returns True if compact success, False otherwise.

void `net_pkt_get_info`(struct k_mem_slab **rx, struct k_mem_slab **tx, struct *net_buf_pool* **rx_data, struct *net_buf_pool* **tx_data)

Get information about predefined RX, TX and DATA pools.

Parameters

- `rx` – Pointer to RX pool is returned.
- `tx` – Pointer to TX pool is returned.
- `rx_data` – Pointer to RX DATA pool is returned.
- `tx_data` – Pointer to TX DATA pool is returned.

struct *net_pkt* *`net_pkt_alloc`(*k_timeout_t* timeout)

Allocate an initialized *net_pkt*.

for the time being, 2 pools are used. One for TX and one for RX. This allocator has to be used for TX.

Parameters

- `timeout` – Maximum time to wait for an allocation.

Returns a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* *`net_pkt_alloc_from_slab`(struct k_mem_slab *slab, *k_timeout_t* timeout)

Allocate an initialized *net_pkt* from a specific slab.

unlike *net_pkt_alloc()* which uses core slabs, this one will use an external slab (see *NET_PKT_SLAB_DEFINE()*). Do *not* use it unless you know what you are doing. Basically, only net_context should be using this, in order to allocate packet and then buffer on its local slab/pool (if any).

Parameters

- `slab` – The slab to use for allocating the packet
- `timeout` – Maximum time to wait for an allocation.

Returns a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* *`net_pkt_rx_alloc`(*k_timeout_t* timeout)

Allocate an initialized *net_pkt* for RX.

for the time being, 2 pools are used. One for TX and one for RX. This allocator has to be used for RX.

Parameters

- `timeout` – Maximum time to wait for an allocation.

Returns a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* *`net_pkt_alloc_on_iface`(struct *net_if* *iface, *k_timeout_t* timeout)

Allocate a network packet for a specific network interface.

Parameters

- `iface` – The network interface the packet is supposed to go through.
- `timeout` – Maximum time to wait for an allocation.

Returns a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* *`net_pkt_rx_alloc_on_iface`(struct *net_if* *iface, *k_timeout_t* timeout)

int `net_pkt_alloc_buffer`(struct *net_pkt* *pkt, size_t size, enum *net_ip_protocol* proto, *k_timeout_t* timeout)

Allocate buffer for a *net_pkt*.

: such allocator will take into account space necessary for headers, MTU, and existing buffer (if any). Beware that, due to all these criteria, the allocated size might be smaller/bigger than requested one.

Parameters

- `pkt` – The network packet requiring buffer to be allocated.
- `size` – The size of buffer being requested.
- `proto` – The IP protocol type (can be 0 for none).
- `timeout` – Maximum time to wait for an allocation.

Returns 0 on success, negative errno code otherwise.

struct *net_pkt* *`net_pkt_alloc_with_buffer`(struct *net_if* *iface, size_t size, *sa_family_t* family, enum *net_ip_protocol* proto, *k_timeout_t* timeout)

Allocate a network packet and buffer at once.

Parameters

- `iface` – The network interface the packet is supposed to go through.
- `size` – The size of buffer.
- `family` – The family to which the packet belongs.
- `proto` – The IP protocol type (can be 0 for none).
- `timeout` – Maximum time to wait for an allocation.

Returns a pointer to a newly allocated *net_pkt* on success, NULL otherwise.

struct *net_pkt* *`net_pkt_rx_alloc_with_buffer`(struct *net_if* *iface, size_t size, *sa_family_t* family, enum *net_ip_protocol* proto, *k_timeout_t* timeout)

void `net_pkt_append_buffer`(struct *net_pkt* *pkt, struct *net_buf* *buffer)

Append a buffer in packet.

Parameters

- `pkt` – Network packet where to append the buffer
- `buffer` – Buffer to append

size_t `net_pkt_available_buffer`(struct *net_pkt* *pkt)

Get available buffer space from a pkt.

Note: Reserved bytes (headroom) in any of the fragments are not considered to be available.

Parameters

- `pkt` – The *net_pkt* which buffer availability should be evaluated

Returns the amount of buffer available

size_t `net_pkt_available_payload_buffer`(struct *net_pkt* *pkt, enum *net_ip_protocol* proto)

Get available buffer space for payload from a pkt.

Unlike *net_pkt_available_buffer()*, this will take into account the headers space.

Note: Reserved bytes (headroom) in any of the fragments are not considered to be available.

Parameters

- `pkt` – The *net_pkt* which payload buffer availability should be evaluated
- `proto` – The IP protocol type (can be 0 for none).

Returns the amount of buffer available for payload

void `net_pkt_trim_buffer`(struct *net_pkt* *pkt)

Trim *net_pkt* buffer.

This will basically check for unused buffers and deallocates them relevantly

Parameters

- `pkt` – The *net_pkt* which buffer will be trimmed

int net_pkt_remove_tail(struct *net_pkt* *pkt, size_t length)

Remove *length* bytes from tail of packet.

This function does not take packet cursor into account. It is a helper to remove unneeded bytes from tail of packet (like appended CRC). It takes care of buffer deallocation if removed bytes span whole buffer(s).

Parameters

- pkt – Network packet
- length – Number of bytes to be removed

Return values

- 0 – On success.
- -EINVAL – If packet length is shorter than *length*.

void net_pkt_cursor_init(struct *net_pkt* *pkt)

Initialize *net_pkt* cursor.

This will initialize the *net_pkt* cursor from its buffer.

Parameters

- pkt – The *net_pkt* whose cursor is going to be initialized

static inline void net_pkt_cursor_backup(struct *net_pkt* *pkt, struct *net_pkt_cursor* *backup)

Backup *net_pkt* cursor.

Parameters

- pkt – The *net_pkt* whose cursor is going to be backed up
- backup – The cursor where to backup *net_pkt* cursor

static inline void net_pkt_cursor_restore(struct *net_pkt* *pkt, struct *net_pkt_cursor* *backup)

Restore *net_pkt* cursor from a backup.

Parameters

- pkt – The *net_pkt* whose cursor is going to be restored
- backup – The cursor from where to restore *net_pkt* cursor

static inline void *net_pkt_cursor_get_pos(struct *net_pkt* *pkt)

Returns current position of the cursor.

Parameters

- pkt – The *net_pkt* whose cursor position is going to be returned

Returns cursor's position

int net_pkt_skip(struct *net_pkt* *pkt, size_t length)

Skip some data from a *net_pkt*.

net_pkt's cursor should be properly initialized Cursor position will be updated after the operation. Depending on the value of pkt->overwrite bit, this function will affect the buffer length or not. If it's true, it will advance the cursor to the requested length. If it's false, it will do the same but if the cursor was already also at the end of existing data, it will increment the buffer length. So in this case, its behavior is just like net_pkt_write or net_pkt_memset, difference being that it will not affect the buffer content itself (which may be just garbage then).

Parameters

- pkt – The *net_pkt* whose cursor will be updated to skip given amount of data from the buffer.
- length – Amount of data to skip in the buffer

Returns 0 in success, negative errno code otherwise.

int `net_pkt_memset`(struct *net_pkt* *pkt, int byte, size_t length)

Memset some data in a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- `pkt` – The *net_pkt* whose buffer to fill starting at the current cursor position.
- `byte` – The byte to write in memory
- `length` – Amount of data to memset with given byte

Returns 0 in success, negative errno code otherwise.

int `net_pkt_copy`(struct *net_pkt* *pkt_dst, struct *net_pkt* *pkt_src, size_t length)

Copy data from a packet into another one.

Both *net_pkt* cursors should be properly initialized and, if needed, positioned using net_pkt_skip. The cursors will be updated after the operation.

Parameters

- `pkt_dst` – Destination network packet.
- `pkt_src` – Source network packet.
- `length` – Length of data to be copied.

Returns 0 on success, negative errno code otherwise.

struct *net_pkt* *`net_pkt_clone`(struct *net_pkt* *pkt, *k_timeout_t* timeout)

Clone pkt and its buffer.

Parameters

- `pkt` – Original pkt to be cloned
- `timeout` – Timeout to wait for free buffer

Returns NULL if error, cloned packet otherwise.

struct *net_pkt* *`net_pkt_shallow_clone`(struct *net_pkt* *pkt, *k_timeout_t* timeout)

Clone pkt and increase the refcount of its buffer.

Parameters

- `pkt` – Original pkt to be shallow cloned
- `timeout` – Timeout to wait for free packet

Returns NULL if error, cloned packet otherwise.

int `net_pkt_read`(struct *net_pkt* *pkt, void *data, size_t length)

Read some data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- `pkt` – The network packet from where to read some data
- `data` – The destination buffer where to copy the data
- `length` – The amount of data to copy

Returns 0 on success, negative errno code otherwise.

static inline int net_pkt_read_u8(struct *net_pkt* *pkt, uint8_t *data)

int net_pkt_read_be16(struct *net_pkt* *pkt, uint16_t *data)

Read uint16_t big endian data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- pkt – The network packet from where to read
- data – The destination uint16_t where to copy the data

Returns 0 on success, negative errno code otherwise.

int net_pkt_read_le16(struct *net_pkt* *pkt, uint16_t *data)

Read uint16_t little endian data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- pkt – The network packet from where to read
- data – The destination uint16_t where to copy the data

Returns 0 on success, negative errno code otherwise.

int net_pkt_read_be32(struct *net_pkt* *pkt, uint32_t *data)

Read uint32_t big endian data from a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- pkt – The network packet from where to read
- data – The destination uint32_t where to copy the data

Returns 0 on success, negative errno code otherwise.

int net_pkt_write(struct *net_pkt* *pkt, const void *data, size_t length)

Write data into a *net_pkt*.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- pkt – The network packet where to write
- data – Data to be written
- length – Length of the data to be written

Returns 0 on success, negative errno code otherwise.

static inline int net_pkt_write_u8(struct *net_pkt* *pkt, uint8_t data)

static inline int net_pkt_write_be16(struct *net_pkt* *pkt, uint16_t data)

static inline int net_pkt_write_be32(struct *net_pkt* *pkt, uint32_t data)

static inline int net_pkt_write_le32(struct *net_pkt* *pkt, uint32_t data)

static inline int net_pkt_write_le16(struct *net_pkt* *pkt, uint16_t data)

size_t net_pkt_remaining_data(struct *net_pkt* *pkt)

Get the amount of data which can be read from current cursor position.

Parameters

- pkt – Network packet

Returns Amount of data which can be read from current pkt cursor

int net_pkt_update_length(struct *net_pkt* *pkt, size_t length)

Update the overall length of a packet.

Unlike *net_pkt_pull()* below, this does not take packet cursor into account. It's mainly a helper dedicated for ipv4 and ipv6 input functions. It shrinks the overall length by given parameter.

Parameters

- pkt – Network packet
- length – The new length of the packet

Returns 0 on success, negative errno code otherwise.

int net_pkt_pull(struct *net_pkt* *pkt, size_t length)

Remove data from the packet at current location.

net_pkt's cursor should be properly initialized and, eventually, properly positioned using net_pkt_skip/read/write. Note that *net_pkt*'s cursor is reset by this function.

Parameters

- pkt – Network packet
- length – Number of bytes to be removed

Returns 0 on success, negative errno code otherwise.

uint16_t net_pkt_get_current_offset(struct *net_pkt* *pkt)

Get the actual offset in the packet from its cursor.

Parameters

- pkt – Network packet.

Returns a valid offset on success, 0 otherwise as there is nothing that can be done to evaluate the offset.

bool net_pkt_is_contiguous(struct *net_pkt* *pkt, size_t size)

Check if a data size could fit contiguously.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip.

Parameters

- pkt – Network packet.
- size – The size to check for contiguity

Returns true if that is the case, false otherwise.

size_t net_pkt_get_contiguous_len(struct *net_pkt* *pkt)

Get the contiguous buffer space

Parameters

- pkt – Network packet

Returns The available contiguous buffer space in bytes starting from the current cursor position. 0 in case of an error.

void *net_pkt_get_data(struct *net_pkt* *pkt, struct *net_pkt_data_access* *access)

Get data from a network packet in a contiguous way.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- pkt – The network packet from where to get the data.
- access – A pointer to a valid *net_pkt_data_access* describing the data to get in a contiguous way.

Returns a pointer to the requested contiguous data, NULL otherwise.

int net_pkt_set_data(struct *net_pkt* *pkt, struct *net_pkt_data_access* *access)

Set contiguous data into a network packet.

net_pkt's cursor should be properly initialized and, if needed, positioned using net_pkt_skip. Cursor position will be updated after the operation.

Parameters

- pkt – The network packet to where the data should be set.
- access – A pointer to a valid *net_pkt_data_access* describing the data to set.

Returns 0 on success, a negative errno otherwise.

static inline int net_pkt_acknowledge_data(struct *net_pkt* *pkt, struct *net_pkt_data_access* *access)

Acknowledge previously contiguous data taken from a network packet Packet needs to be set to overwrite mode.

struct net_pkt_cursor

#include <net_pkt.h>

Public Members

struct *net_buf* *buf

Current *net_buf* pointer by the cursor

uint8_t *pos

Current position in the data buffer of the *net_buf*

struct net_pkt

#include <net_pkt.h> Network packet.

Note that if you add new fields into *net_pkt*, remember to update *net_pkt_clone()* function.

Public Members

intptr_t fifo

The fifo is used by RX/TX threads and by socket layer. The *net_pkt* is queued via fifo to the processing thread.

struct k_mem_slab *slab

Slab pointer from where it belongs to

union *net_pkt*.[anonymous] `[anonymous]`

buffer holding the packet

struct *net_pkt_cursor* `cursor`

Internal buffer iterator used for reading/writing

struct net_context `*context`

Network connection context

struct *net_if* `*iface`

Network interface

struct `net_pkt_data_access`

#include <net_pkt.h>

7.20.3 Networking Technologies

Ethernet

- *Overview*
- *API Reference*

Virtual LAN (VLAN) Support

- *Overview*
- *API Reference*

Overview Virtual LAN (VLAN) is a partitioned and isolated computer network at the data link layer (OSI layer 2). For ethernet network this refers to IEEE 802.1Q

In Zephyr, each individual VLAN is modeled as a virtual network interface. This means that there is an ethernet network interface that corresponds to a real physical ethernet port in the system. A virtual network interface is created for each VLAN, and this virtual network interface connects to the real network interface. This is similar to how Linux implements VLANs. The *eth0* is the real network interface and *vlan0* is a virtual network interface that is run on top of *eth0*.

VLAN support must be enabled at compile time by setting option `CONFIG_NET_VLAN` and `CONFIG_NET_VLAN_COUNT` to reflect how many network interfaces there will be in the system. For example, if there is one network interface without VLAN support, and two with VLAN support, the `CONFIG_NET_VLAN_COUNT` option should be set to 3.

Even if VLAN is enabled in a `prj.conf` file, the VLAN needs to be activated at runtime by the application. The VLAN API provides a *`net_eth_vlan_enable()`* function to do that. The application needs to give the network interface and desired VLAN tag as a parameter to that function. The VLAN tagging for a given network interface can be disabled by a *`net_eth_vlan_disable()`* function. The application needs to configure the VLAN network interface itself, such as setting the IP address, etc.

See also the VLAN sample application for API usage example. The source code for that sample application can be found at samples/net/vlan.

The net-shell module contains *net vlan add* and *net vlan del* commands that can be used to enable or disable VLAN tags for a given network interface.

See the IEEE 802.1Q spec for more information about ethernet VLANs.

API Reference

group `vlan_api`

VLAN definitions and helpers.

Defines

`NET_VLAN_TAG_UNSPEC`

Unspecified VLAN tag value

Functions

static inline uint16_t `net_eth_vlan_get_vid`(uint16_t tci)

Get VLAN identifier from TCI.

Parameters

- `tci` – VLAN tag control information.

Returns VLAN identifier.

static inline uint8_t `net_eth_vlan_get_dei`(uint16_t tci)

Get Drop Eligible Indicator from TCI.

Parameters

- `tci` – VLAN tag control information.

Returns Drop eligible indicator.

static inline uint8_t `net_eth_vlan_get_pcp`(uint16_t tci)

Get Priority Code Point from TCI.

Parameters

- `tci` – VLAN tag control information.

Returns Priority code point.

static inline uint16_t `net_eth_vlan_set_vid`(uint16_t tci, uint16_t vid)

Set VLAN identifier to TCI.

Parameters

- `tci` – VLAN tag control information.
- `vid` – VLAN identifier.

Returns New TCI value.

static inline uint16_t `net_eth_vlan_set_dei`(uint16_t tci, bool dei)

Set Drop Eligible Indicator to TCI.

Parameters

- `tci` – VLAN tag control information.
- `dei` – Drop eligible indicator.

Returns New TCI value.

static inline uint16_t `net_eth_vlan_set_pcp`(uint16_t tci, uint8_t pcp)

Set Priority Code Point to TCI.

Parameters

- `tci` – VLAN tag control information.
- `pcp` – Priority code point.

Returns New TCI value.

Link Layer Discovery Protocol

Overview The Link Layer Discovery Protocol (LLDP) is a vendor-neutral link layer protocol used by network devices for advertising their identity, capabilities, and neighbors on a wired Ethernet network.

For more information, see this LLDP Wikipedia article.

API Reference

group `lldp`

LLDP definitions and helpers.

Defines

`net_lldp_set_lldpdu`(iface)

Set LLDP protocol data unit (LLDPDU) for the network interface.

Parameters

- `iface` – Network interface

Returns <0 if error, index in lldp array if iface is found there

`net_lldp_unset_lldpdu`(iface)

Unset LLDP protocol data unit (LLDPDU) for the network interface.

Parameters

- `iface` – Network interface

Typedefs

typedef enum *net_verdict* (*`net_lldp_recv_cb_t`)(struct *net_if* *iface, struct *net_pkt* *pkt)

LLDP Receive packet callback.

Callback gets called upon receiving packet. It is responsible for freeing packet or indicating to the stack that it needs to free packet by returning correct net_verdict.

Returns:

- NET_DROP, if packet was invalid, rejected or we want the stack to free it. In this case the core stack will free the packet.

- NET_OK, if the packet was accepted, in this case the ownership of the *net_pkt* goes to callback and core network stack will forget it.

Enums

enum `net_lldp_tlv_type`

TLV Types. Please refer to table 8-1 from IEEE 802.1AB standard.

Values:

enumerator `LLDP_TLV_END_LLDPDU` = 0

End Of LLDPDU (optional)

enumerator `LLDP_TLV_CHASSIS_ID` = 1

Chassis ID (mandatory)

enumerator `LLDP_TLV_PORT_ID` = 2

Port ID (mandatory)

enumerator `LLDP_TLV_TTL` = 3

Time To Live (mandatory)

enumerator `LLDP_TLV_PORT_DESC` = 4

Port Description (optional)

enumerator `LLDP_TLV_SYSTEM_NAME` = 5

System Name (optional)

enumerator `LLDP_TLV_SYSTEM_DESC` = 6

System Description (optional)

enumerator `LLDP_TLV_SYSTEM_CAPABILITIES` = 7

System Capability (optional)

enumerator `LLDP_TLV_MANAGEMENT_ADDR` = 8

Management Address (optional)

enumerator `LLDP_TLV_ORG_SPECIFIC` = 127

Org specific TLVs (optional)

Functions

int `net_lldp_config`(struct *net_if* *iface, const struct *net_lldpdu* *lldpdu)

Set the LLDP data unit for a network interface.

Parameters

- `iface` – Network interface
- `lldpdu` – LLDP data unit struct

Returns 0 if ok, <0 if error

int net_lldp_config_optional(struct *net_if* *iface, const uint8_t *tlv, size_t len)

Set the Optional LLDP TLVs for a network interface.

Parameters

- iface – Network interface
- tlv – LLDP optional TLVs following mandatory part
- len – Length of the optional TLVs

Returns 0 if ok, <0 if error

void net_lldp_init(void)

Initialize LLDP engine.

int net_lldp_register_callback(struct *net_if* *iface, *net_lldp_recv_cb_t* cb)

Register LLDP Rx callback function.

Parameters

- iface – Network interface
- cb – Callback function

Returns 0 if ok, < 0 if error

enum *net_verdict* net_lldp_recv(struct *net_if* *iface, struct *net_pkt* *pkt)

Parse LLDP packet.

Parameters

- iface – Network interface
- pkt – Network packet

Returns Return the policy for network buffer

struct net_lldp_chassis_tlv

#include <lldp.h> Chassis ID TLV, see chapter 8.5.2 in IEEE 802.1AB

Public Members

uint16_t type_length

7 bits for type, 9 bits for length

uint8_t subtype

ID subtype

uint8_t value[NET_LLDP_CHASSIS_ID_VALUE_LEN]

Chassis ID value

struct net_lldp_port_tlv

#include <lldp.h> Port ID TLV, see chapter 8.5.3 in IEEE 802.1AB

Public Members

uint16_t type_length

7 bits for type, 9 bits for length

uint8_t `subtype`

ID subtype

uint8_t `value`[NET_LLDP_PORT_ID_VALUE_LEN]

Port ID value

struct `net_lldp_time_to_live_tlv`

#include <lldp.h> Time To Live TLV, see chapter 8.5.4 in IEEE 802.1AB

Public Members

uint16_t `type_length`

7 bits for type, 9 bits for length

uint16_t `ttl`

Time To Live (TTL) value

struct `net_lldpdu`

#include <lldp.h> LLDP Data Unit (LLDPDU) shall contain the following ordered TLVs as stated in "8.2 LLDPDU format" from the IEEE 802.1AB

Public Members

struct *net_lldp_chassis_tlv* `chassis_id`

Mandatory Chassis TLV

struct *net_lldp_port_tlv* `port_id`

Mandatory Port TLV

struct *net_lldp_time_to_live_tlv* `ttl`

Mandatory TTL TLV

IEEE 802.1Qav

Overview Credit-based shaping is an alternative scheduling algorithm used in network schedulers to achieve fairness when sharing a limited network resource. Zephyr has support for configuring a credit-based shaper described in the IEEE 802.1Qav-2009 standard. Zephyr does not implement the actual shaper; it only provides a way to configure the shaper implemented by the Ethernet device driver.

Enabling 802.1Qav To enable 802.1Qav shaper, the Ethernet device driver must declare that it supports credit-based shaping. The Ethernet driver's capability function must return `ETHERNET_QAV` value for this purpose. Typically also priority queues `ETHERNET_PRIORITY_QUEUES` need to be supported.

```
static enum ethernet_hw_caps eth_get_capabilities(const struct device *dev)
{
        ARG_UNUSED(dev);
```

(continues on next page)

(continued from previous page)

```
        return ETHERNET_QAV | ETHERNET_PRIORITY_QUEUES |
               ETHERNET_HW_VLAN | ETHERNET_LINK_10BASE_T |
               ETHERNET_LINK_100BASE_T;
}
```

See `sam-e70-xplained` board Ethernet driver drivers/ethernet/eth_sam_gmac.c for an example.

Configuring 802.1Qav The application can configure the credit-based shaper like this:

```
#include <net/net_if.h>
#include <net/ethernet.h>
#include <net/ethernet_mgmt.h>

static void qav_set_status(struct net_if *iface,
                           int queue_id, bool enable)
{
        struct ethernet_req_params params;
        int ret;

        memset(&params, 0, sizeof(params));

        params.qav_param.queue_id = queue_id;
        params.qav_param.enabled = enable;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_STATUS;

        /* Disable or enable Qav for a queue */
        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot %s Qav for queue %d for interface %p",
                        enable ? "enable" : "disable",
                        queue_id, iface);
        }
}

static void qav_set_bandwidth_and_slope(struct net_if *iface,
                                        int queue_id,
                                        unsigned int bandwidth,
                                        unsigned int idle_slope)
{
        struct ethernet_req_params params;
        int ret;

        memset(&params, 0, sizeof(params));

        params.qav_param.queue_id = queue_id;
        params.qav_param.delta_bandwidth = bandwidth;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_DELTA_BANDWIDTH;

        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot set Qav delta bandwidth %u for "
                        "queue %d for interface %p",
```

(continues on next page)

(continued from previous page)

```
                        bandwidth, queue_id, iface);
        }

        params.qav_param.idle_slope = idle_slope;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_IDLE_SLOPE;

        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot set Qav idle slope %u for "
                        "queue %d for interface %p",
                        idle_slope, queue_id, iface);
        }
}
```

Overview Ethernet is a networking technology commonly used in local area networks (LAN). For more information, see this Ethernet Wikipedia article.

Zephyr supports following Ethernet features:

- 10, 100 and 1000 Mbit/sec links
- Auto negotiation
- Half/full duplex
- Promiscuous mode
- TX and RX checksum offloading
- MAC address filtering
- *Virtual LANs*
- *Priority queues*
- *IEEE 802.1AS (gPTP)*
- *IEEE 802.1Qav (credit based shaping)*
- *LLDP (Link Layer Discovery Protocol)*

Not all Ethernet device drivers support all of these features. You can see what is supported by `net iface` net-shell command. It will print currently supported Ethernet features.

API Reference

group `ethernet`

Ethernet support functions.

Defines

`ETH_NET_DEVICE_INIT(dev_name, drv_name, init_fn, pm_control_fn, data, cfg, prio, api, mtu)`

Create an Ethernet network interface and bind it to network device.

Parameters

- `dev_name` – Network device name.
- `drv_name` – The name this instance of the driver exposes to the system.

- `init_fn` – Address to the init function of the driver.
- `pm_control_fn` – Pointer to pm_control function. Can be NULL if not implemented.
- `data` – Pointer to the device's private data.
- `cfg` – The address to the structure containing the configuration information for this instance of the driver.
- `prio` – The initialization level at which configuration occurs.
- `api` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- `mtu` – Maximum transfer unit in bytes for this network interface.

`ETH_NET_DEVICE_DT_DEFINE(node_id, init_fn, pm_control_fn, data, cfg, prio, api, mtu)`

Like ETH_NET_DEVICE_INIT but taking metadata from a devicetree. Create an Ethernet network interface and bind it to network device.

Parameters

- `node_id` – The devicetree node identifier.
- `init_fn` – Address to the init function of the driver.
- `pm_control_fn` – Pointer to pm_control function. Can be NULL if not implemented.
- `data` – Pointer to the device's private data.
- `cfg` – The address to the structure containing the configuration information for this instance of the driver.
- `prio` – The initialization level at which configuration occurs.
- `api` – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- `mtu` – Maximum transfer unit in bytes for this network interface.

`ETH_NET_DEVICE_DT_INST_DEFINE(inst, ...)`

Like ETH_NET_DEVICE_DT_DEFINE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- `inst` – instance number. This is replaced by `DT_DRV_COMPAT(inst)` in the call to ETH_NET_DEVICE_DT_DEFINE.
- `...` – other parameters as expected by ETH_NET_DEVICE_DT_DEFINE.

Enums

enum `ethernet_hw_caps`

Ethernet hardware capabilities

Values:

enumerator `ETHERNET_HW_TX_CHKSUM_OFFLOAD` = *BIT*(0)

TX Checksum offloading supported for all of IPv4, UDP, TCP

enumerator `ETHERNET_HW_RX_CHKSUM_OFFLOAD` = *BIT*(1)

RX Checksum offloading supported for all of IPv4, UDP, TCP

enumerator ETHERNET_HW_VLAN = *BIT*(2)
VLAN supported

enumerator ETHERNET_AUTO_NEGOTIATION_SET = *BIT*(3)
Enabling/disabling auto negotiation supported

enumerator ETHERNET_LINK_10BASE_T = *BIT*(4)
10 Mbits link supported

enumerator ETHERNET_LINK_100BASE_T = *BIT*(5)
100 Mbits link supported

enumerator ETHERNET_LINK_1000BASE_T = *BIT*(6)
1 Gbits link supported

enumerator ETHERNET_DUPLEX_SET = *BIT*(7)
Changing duplex (half/full) supported

enumerator ETHERNET_PTP = *BIT*(8)
IEEE 802.1AS (gPTP) clock supported

enumerator ETHERNET_QAV = *BIT*(9)
IEEE 802.1Qav (credit-based shaping) supported

enumerator ETHERNET_PROMISC_MODE = *BIT*(10)
Promiscuous mode supported

enumerator ETHERNET_PRIORITY_QUEUES = *BIT*(11)
Priority queues available

enumerator ETHERNET_HW_FILTERING = *BIT*(12)
MAC address filtering supported

enumerator ETHERNET_LLDP = *BIT*(13)
Link Layer Discovery Protocol supported

enumerator ETHERNET_HW_VLAN_TAG_STRIP = *BIT*(14)
VLAN Tag stripping

enumerator ETHERNET_DSA_SLAVE_PORT = *BIT*(15)
DSA switch

enumerator ETHERNET_DSA_MASTER_PORT = *BIT*(16)

enumerator ETHERNET_QBV = *BIT*(17)
IEEE 802.1Qbv (scheduled traffic) supported

enumerator ETHERNET_QBU = *BIT*(18)
IEEE 802.1Qbu (frame preemption) supported

enumerator ETHERNET_TXTIME = *BIT*(19)

TXTIME supported

enum ethernet_flags

Values:

enumerator ETH_CARRIER_UP

Functions

void ethernet_init(struct *net_if* *iface)

Initialize Ethernet L2 stack for a given interface.

Parameters

- iface – A valid pointer to a network interface

void net_eth_ipv4_mcast_to_mac_addr(const struct *in_addr* *ipv4_addr, struct net_eth_addr *mac_addr)

Convert IPv4 multicast address to Ethernet address.

Parameters

- ipv4_addr – IPv4 multicast address
- mac_addr – Output buffer for Ethernet address

void net_eth_ipv6_mcast_to_mac_addr(const struct *in6_addr* *ipv6_addr, struct net_eth_addr *mac_addr)

Convert IPv6 multicast address to Ethernet address.

Parameters

- ipv6_addr – IPv6 multicast address
- mac_addr – Output buffer for Ethernet address

static inline enum *ethernet_hw_caps* net_eth_get_hw_capabilities(struct *net_if* *iface)

Return ethernet device hardware capability information.

Parameters

- iface – Network interface

Returns Hardware capabilities

static inline int net_eth_vlan_enable(struct *net_if* *iface, uint16_t tag)

Add VLAN tag to the interface.

Parameters

- iface – Interface to use.
- tag – VLAN tag to add

Returns 0 if ok, <0 if error

static inline int net_eth_vlan_disable(struct *net_if* *iface, uint16_t tag)

Remove VLAN tag from the interface.

Parameters

- iface – Interface to use.
- tag – VLAN tag to remove

Returns 0 if ok, <0 if error

static inline uint16_t `net_eth_get_vlan_tag`(struct *net_if* *iface)

Return VLAN tag specified to network interface.

Parameters

- `iface` – Network interface.

Returns VLAN tag for this interface or NET_VLAN_TAG_UNSPEC if VLAN is not configured for that interface.

static inline struct *net_if* *`net_eth_get_vlan_iface`(struct *net_if* *iface, uint16_t tag)

Return network interface related to this VLAN tag.

Parameters

- `iface` – Master network interface. This is used to get the pointer to ethernet L2 context
- `tag` – VLAN tag

Returns Network interface related to this tag or NULL if no such interface exists.

static inline bool `net_eth_is_vlan_enabled`(struct *ethernet_context* *ctx, struct *net_if* *iface)

Check if VLAN is enabled for a specific network interface.

Parameters

- `ctx` – Ethernet context
- `iface` – Network interface

Returns True if VLAN is enabled for this network interface, false if not.

static inline bool `net_eth_get_vlan_status`(struct *net_if* *iface)

Get VLAN status for a given network interface (enabled or not).

Parameters

- `iface` – Network interface

Returns True if VLAN is enabled for this network interface, false if not.

void `net_eth_carrier_on`(struct *net_if* *iface)

Inform ethernet L2 driver that ethernet carrier is detected. This happens when cable is connected.

Parameters

- `iface` – Network interface

void `net_eth_carrier_off`(struct *net_if* *iface)

Inform ethernet L2 driver that ethernet carrier was lost. This happens when cable is disconnected.

Parameters

- `iface` – Network interface

int `net_eth_promisc_mode`(struct *net_if* *iface, bool enable)

Set promiscuous mode either ON or OFF.

Parameters

- `iface` – Network interface
- `enable` – on (true) or off (false)

Returns 0 if mode set or unset was successful, <0 otherwise.

static inline const struct *device* *net_eth_get_ptp_clock(struct *net_if* *iface)

Return PTP clock that is tied to this ethernet network interface.

Parameters

- iface – Network interface

Returns Pointer to PTP clock if found, NULL if not found or if this ethernet interface does not support PTP.

const struct *device* *net_eth_get_ptp_clock_by_index(int index)

Return PTP clock that is tied to this ethernet network interface index.

Parameters

- index – Network interface index

Returns Pointer to PTP clock if found, NULL if not found or if this ethernet interface index does not support PTP.

static inline int net_eth_get_ptp_port(struct *net_if* *iface)

Return gPTP port number attached to this interface.

Parameters

- iface – Network interface

Returns Port number, no such port if < 0

struct ethernet_qav_param

#include <ethernet.h>

Public Members

int queue_id

ID of the priority queue to use

enum ethernet_qav_param_type type

Type of Qav parameter

bool enabled

True if Qav is enabled for queue

unsigned int delta_bandwidth

Delta Bandwidth (percentage of bandwidth)

unsigned int idle_slope

Idle Slope (bits per second)

unsigned int oper_idle_slope

Oper Idle Slope (bits per second)

unsigned int traffic_class

Traffic class the queue is bound to

struct ethernet_qbv_param

#include <ethernet.h>

Public Members

int `port_id`
Port id

enum ethernet_qbv_param_type `type`
Type of Qbv parameter

enum ethernet_qbv_state_type `state`
What state (Admin/Oper) parameters are these

bool `enabled`
True if Qbv is enabled or not

bool `gate_status`[NET_TC_TX_COUNT]
True = open, False = closed

enum ethernet_gate_state_operation `operation`
GateState operation

uint32_t `time_interval`
Time interval ticks (nanoseconds)

uint16_t `row`
Gate control list row

uint32_t `gate_control_list_len`
Number of entries in gate control list

struct *net_ptp_extended_time* `base_time`
Base time

struct *net_ptp_time* `cycle_time`
Cycle time

uint32_t `extension_time`
Extension time (nanoseconds)

struct `ethernet_qbu_param`
#include <ethernet.h>

Public Members

int `port_id`
Port id

enum ethernet_qbu_param_type `type`
Type of Qbu parameter

uint32_t `hold_advance`

Hold advance (nanoseconds)

uint32_t `release_advance`

Release advance (nanoseconds)

enum ethernet_qbu_preempt_status `frame_preempt_statuses`[NET_TC_TX_COUNT]

sequence of framePreemptionAdminStatus values.

bool `enabled`

True if Qbu is enabled or not

bool `link_partner_status`

Link partner status (from Qbr)

uint8_t `additional_fragment_size`

Additional fragment size (from Qbr). The minimum non-final fragment size is (additional_fragment_size + 1) * 64 octets

struct `ethernet_filter`

#include <ethernet.h>

Public Members

enum ethernet_filter_type `type`

Type of filter

struct net_eth_addr `mac_address`

MAC address to filter

bool `set`

Set (true) or unset (false) the filter

struct `ethernet_txtime_param`

#include <ethernet.h>

Public Members

enum ethernet_txtime_param_type `type`

Type of TXTIME parameter

int `queue_id`

Queue number for configuring TXTIME

bool `enable_txtime`

Enable or disable TXTIME per queue

struct `ethernet_api`

#include <ethernet.h>

Public Members

struct net_if_api `iface_api`

The net_if_api must be placed in first position in this struct so that we are compatible with network interface API.

int (*`start`)(const struct *device* *dev)

Start the device

int (*`stop`)(const struct *device* *dev)

Stop the device

enum *ethernet_hw_caps* (*`get_capabilities`)(const struct *device* *dev)

Get the device capabilities

int (*`set_config`)(const struct *device* *dev, enum ethernet_config_type type, const struct ethernet_config *config)

Set specific hardware configuration

int (*`get_config`)(const struct *device* *dev, enum ethernet_config_type type, struct ethernet_config *config)

Get hardware specific configuration

int (*`send`)(const struct *device* *dev, struct *net_pkt* *pkt)

Send a network packet

struct `ethernet_context`

#include <ethernet.h> Ethernet L2 context that is needed for VLAN

Public Members

atomic_t `flags`

Flags representing ethernet state, which are accessed from multiple threads.

struct *k_work* `carrier_work`

Carrier ON/OFF handler worker. This is used to create network interface UP/DOWN event when ethernet L2 driver notices carrier ON/OFF situation. We must not create another network management event from inside management handler thus we use worker thread to trigger the UP/DOWN event.

struct *net_if* *`iface`

Network interface.

enum *net_l2_flags* `ethernet_l2_flags`

This tells what L2 features does ethernet support.

bool `is_net_carrier_up`
: Is network carrier up

bool `is_init`
: Is this context already initialized

group `ethernet_mii`

Ethernet MII (media independent interface) functions.

Defines

`MII_BMCR`
: Basic Mode Control Register

`MII_BMSR`
: Basic Mode Status Register

`MII_PHYID1R`
: PHY ID 1 Register

`MII_PHYID2R`
: PHY ID 2 Register

`MII_ANAR`
: Auto-Negotiation Advertisement Register

`MII_ANLPAR`
: Auto-Negotiation Link Partner Ability Reg

`MII_ANER`
: Auto-Negotiation Expansion Register

`MII_ANNPTR`
: Auto-Negotiation Next Page Transmit Register

`MII_ANLPRNPR`
: Auto-Negotiation Link Partner Received Next Page Reg

`MII_MMD_ACR`
: MMD Access Control Register

`MII_MMD_AADR`
: MMD Access Address Data Register

`MII_ESTAT`
: Extended Status Register

MII_BMCR_RESET
: PHY reset

MII_BMCR_LOOPBACK
: enable loopback mode

MII_BMCR_SPEED_LSB
: 10=1000Mbps 01=100Mbps; 00=10Mbps

MII_BMCR_AUTONEG_ENABLE
: Auto-Negotiation enable

MII_BMCR_POWER_DOWN
: power down mode

MII_BMCR_ISOLATE
: isolate electrically PHY from MII

MII_BMCR_AUTONEG_RESTART
: restart auto-negotiation

MII_BMCR_DUPLEX_MODE
: full duplex mode

MII_BMCR_SPEED_MSB
: 10=1000Mbps 01=100Mbps; 00=10Mbps

MII_BMCR_SPEED_MASK
: Link Speed Field

MII_BMCR_SPEED_10
: select speed 10 Mb/s

MII_BMCR_SPEED_100
: select speed 100 Mb/s

MII_BMCR_SPEED_1000
: select speed 1000 Mb/s

MII_BMSR_100BASE_T4
: 100BASE-T4 capable

MII_BMSR_100BASE_X_FULL
: 100BASE-X full duplex capable

MII_BMSR_100BASE_X_HALF
: 100BASE-X half duplex capable

MII_BMSR_10_FULL
: 10 Mb/s full duplex capable

MII_BMSR_10_HALF
: 10 Mb/s half duplex capable

MII_BMSR_100BASE_T2_FULL
: 100BASE-T2 full duplex capable

MII_BMSR_100BASE_T2_HALF
: 100BASE-T2 half duplex capable

MII_BMSR_EXTEND_STATUS
: extend status information in reg 15

MII_BMSR_MF_PREAMB_SUPPR
: PHY accepts management frames with preamble suppressed

MII_BMSR_AUTONEG_COMPLETE
: Auto-negotiation process completed

MII_BMSR_REMOTE_FAULT
: remote fault detected

MII_BMSR_AUTONEG_ABILITY
: PHY is able to perform Auto-Negotiation

MII_BMSR_LINK_STATUS
: link is up

MII_BMSR_JABBER_DETECT
: jabber condition detected

MII_BMSR_EXTEND_CAPAB
: extended register capabilities

MII_ADVERTISE_NEXT_PAGE
: next page

MII_ADVERTISE_LPACK
: link partner acknowledge response

MII_ADVERTISE_REMOTE_FAULT
: remote fault

MII_ADVERTISE_ASYM_PAUSE
: try for asymmetric pause

`MII_ADVERTISE_PAUSE`
: try for pause

`MII_ADVERTISE_100BASE_T4`
: try for 100BASE-T4 support

`MII_ADVERTISE_100_FULL`
: try for 100BASE-X full duplex support

`MII_ADVERTISE_100_HALF`
: try for 100BASE-X support

`MII_ADVERTISE_10_FULL`
: try for 10 Mb/s full duplex support

`MII_ADVERTISE_10_HALF`
: try for 10 Mb/s half duplex support

`MII_ADVERTISE_SEL_MASK`
: Selector Field

`MII_ADVERTISE_SEL_IEEE_802_3`

`MII_ADVERTISE_ALL`

IEEE 802.15.4

Overview IEEE 802.15.4 is a technical standard which defines the operation of low-rate wireless personal area networks (LR-WPANs). For more detailed overview of this standard, see this IEEE 802.15.4 Wikipedia article. Also, see IEEE GET Program for creating an IEEE account and downloading the specification.

Zephyr supports IEEE 802.15.4 with Thread and 6LoWPAN. The Thread implementation is based on OpenThread. The IPv6 header compression in 6LoWPAN is shared with the Bluetooth IPSP (IP support profile).

API Reference

IEEE 802.15.4

group `ieee802154`
: IEEE 802.15.4 library.

Defines

IEEE802154_MAX_ADDR_LENGTH

IEEE802154_NO_CHANNEL

IEEE802154_L2_CTX_TYPE

IEEE802154_AR_FLAG_SET

Typedefs

typedef void (*energy_scan_done_cb_t)(const struct *device* *dev, int16_t max_ed)

typedef void (*ieee802154_event_cb_t)(const struct *device* *dev, enum *ieee802154_event* evt, void *event_params)

Enums

enum ieee802154_channel

IEEE 802.15.4 Channel assignments.

Channel numbering for 868 MHz, 915 MHz, and 2450 MHz bands.

- Channel 0 is for 868.3 MHz.
- Channels 1-10 are for 906 to 924 MHz with 2 MHz channel spacing.
- Channels 11-26 are for 2405 to 2530 MHz with 5 MHz channel spacing.

For more information, please refer to 802.15.4-2015 Section 10.1.2.2.

Values:

enumerator IEEE802154_SUB_GHZ_CHANNEL_MIN = 0

enumerator IEEE802154_SUB_GHZ_CHANNEL_MAX = 10

enumerator IEEE802154_2_4_GHZ_CHANNEL_MIN = 11

enumerator IEEE802154_2_4_GHZ_CHANNEL_MAX = 26

enum ieee802154_hw_caps

Values:

enumerator IEEE802154_HW_FCS = *BIT*(0)

enumerator IEEE802154_HW_PROMISC = *BIT*(1)

enumerator IEEE802154_HW_FILTER = *BIT*(2)

enumerator IEEE802154_HW_CSMA = *BIT*(3)

enumerator IEEE802154_HW_2_4_GHZ = *BIT*(4)

enumerator IEEE802154_HW_TX_RX_ACK = *BIT*(5)

enumerator IEEE802154_HW_SUB_GHZ = *BIT*(6)

enumerator IEEE802154_HW_ENERGY_SCAN = *BIT*(7)

enumerator IEEE802154_HW_TXTIME = *BIT*(8)

enumerator IEEE802154_HW_SLEEP_TO_TX = *BIT*(9)

enumerator IEEE802154_HW_TX_SEC = *BIT*(10)

enumerator IEEE802154_HW_RXTIME = *BIT*(11)

enum ieee802154_filter_type

Values:

enumerator IEEE802154_FILTER_TYPE_IEEE_ADDR

enumerator IEEE802154_FILTER_TYPE_SHORT_ADDR

enumerator IEEE802154_FILTER_TYPE_PAN_ID

enumerator IEEE802154_FILTER_TYPE_SRC_IEEE_ADDR

enumerator IEEE802154_FILTER_TYPE_SRC_SHORT_ADDR

enum ieee802154_event

Values:

enumerator IEEE802154_EVENT_TX_STARTED

enumerator IEEE802154_EVENT_RX_FAILED

enumerator IEEE802154_EVENT_SLEEP

enum ieee802154_rx_fail_reason

Values:

enumerator IEEE802154_RX_FAIL_NOT_RECEIVED

enumerator IEEE802154_RX_FAIL_INVALID_FCS

enumerator IEEE802154_RX_FAIL_ADDR_FILTERED

enumerator IEEE802154_RX_FAIL_OTHER

enum ieee802154_tx_mode

IEEE802.15.4 Transmission mode.

Values:

enumerator IEEE802154_TX_MODE_DIRECT

Transmit packet immediately, no CCA.

enumerator IEEE802154_TX_MODE_CCA

Perform CCA before packet transmission.

enumerator IEEE802154_TX_MODE_CSMA_CA

Perform full CSMA CA procedure before packet transmission.

enumerator IEEE802154_TX_MODE_TXTIME

Transmit packet in the future, at specified time, no CCA.

enumerator IEEE802154_TX_MODE_TXTIME_CCA

Transmit packet in the future, perform CCA before transmission.

enum ieee802154_fpb_mode

IEEE802.15.4 Frame Pending Bit table address matching mode.

Values:

enumerator IEEE802154_FPB_ADDR_MATCH_THREAD

The pending bit shall be set only for addresses found in the list.

enumerator IEEE802154_FPB_ADDR_MATCH_ZIGBEE

The pending bit shall be cleared for short addresses found in the list.

enum ieee802154_config_type

IEEE802.15.4 driver configuration types.

Values:

enumerator IEEE802154_CONFIG_AUTO_ACK_FPB

Indicates how radio driver should set Frame Pending bit in ACK responses for Data Requests. If enabled, radio driver should determine whether to set the bit or not based on the information provided with IEEE802154_CONFIG_ACK_FPB config and FPB address matching mode specified. Otherwise, Frame Pending bit should be set to 1(see IEEE Std 802.15.4-2006, 7.2.2.3.1).

enumerator IEEE802154_CONFIG_ACK_FPB

Indicates whether to set ACK Frame Pending bit for specific address or not. Disabling the Frame Pending bit with no address provided (NULL pointer) should disable it for all enabled addresses.

enumerator IEEE802154_CONFIG_PAN_COORDINATOR

Indicates whether the device is a PAN coordinator.

enumerator IEEE802154_CONFIG_PROMISCUOUS

Enable/disable promiscuous mode.

enumerator IEEE802154_CONFIG_EVENT_HANDLER

Specifies new radio event handler. Specifying NULL as a handler will disable radio events notification.

enumerator IEEE802154_CONFIG_MAC_KEYS

Updates MAC keys and key index for radios supporting transmit security.

enumerator IEEE802154_CONFIG_FRAME_COUNTER

Sets the current MAC frame counter value for radios supporting transmit security.

enumerator IEEE802154_CONFIG_RX_SLOT

Configure a radio reception slot. This can be used for any scheduler reception, e.g.: Zigbee GP device, CSL, TSCH, etc.

In order to configure a CSL receiver the upper layer should combine several configuration options in the following way:

i. Use IEEE802154_CONFIG_ENH_ACK_HEADER_IE once to inform the radio driver of the short and extended addresses of the peer to which it should inject CSL IEs.
ii. Use IEEE802154_CONFIG_CSL_RX_TIME periodically, before each use of IEEE802154_CONFIG_CSL_PERIOD setting parameters of the nearest CSL RX window, and before each use of IEEE_CONFIG_RX_SLOT setting parameters of the following (not the nearest one) CSL RX window, to allow the radio driver to calculate the proper CSL Phase to the nearest CSL window to inject in the CSL IEs for both transmitted data and ack frames.
iii. Use IEEE802154_CONFIG_CSL_PERIOD on each value change to update the current CSL period value which will be injected in the CSL IEs together with the CSL Phase based on IEEE802154_CONFIG_CSL_RX_TIME.
iv. Use IEEE802154_CONFIG_RX_SLOT periodically to schedule the immediate receive window earlier enough before the expected window start time, taking into account possible clock drifts and scheduling uncertainties.

This diagram shows the usage of the four options over time: Start CSL Schedule CSL window

ENH_ACK_HEADER_IE CSL_RX_TIME (following window) | | | CSL_RX_TIME (nearest window) | RX_SLOT (nearest window) | | | | | | CSL_PERIOD | | | | | | | v v v v v v ——————————————————————————————-—[CSL window]–—+ ^ | | | +——————————— loop ———+

enumerator IEEE802154_CONFIG_CSL_PERIOD

Configure CSL receiver (Endpoint) period

enumerator IEEE802154_CONFIG_CSL_RX_TIME

Configure the next CSL receive window center, in units of microseconds, based on the radio time.

enumerator `IEEE802154_CONFIG_ENH_ACK_HEADER_IE`

Indicates whether to inject IE into ENH ACK Frame for specific address or not. Disabling the ENH ACK with no address provided (NULL pointer) should disable it for all enabled addresses.

Functions

static inline bool `ieee802154_is_ar_flag_set`(struct *net_buf* *frag)

Check if AR flag is set on the frame inside given *net_pkt*.

Parameters

- `frag` – A valid pointer on a *net_buf* structure, must not be NULL, and its length should be at least made of 1 byte (ACK frames are the smallest frames on 15.4 and made of 3 bytes, not not counting the FCS part).

Returns True if AR flag is set, False otherwise

enum *net_verdict* `ieee802154_radio_handle_ack`(struct *net_if* *iface, struct *net_pkt* *pkt)

Radio driver ACK handling function that hw drivers should use.

ACK handling requires fast handling and thus such function helps to hook directly the hw drivers to the radio driver.

Parameters

- `iface` – A valid pointer on a network interface that received the packet
- `pkt` – A valid pointer on a packet to check

Returns NET_OK if it was handled, NET_CONTINUE otherwise

void `ieee802154_init`(struct *net_if* *iface)

Initialize L2 stack for a given interface.

Parameters

- `iface` – A valid pointer on a network interface

struct `ieee802154_security_ctx`

#include <ieee802154.h>

struct `ieee802154_context`

#include <ieee802154.h>

struct `ieee802154_filter`

#include <ieee802154_radio.h>

struct `ieee802154_key`

#include <ieee802154_radio.h>

struct `ieee802154_config`

#include <ieee802154_radio.h> IEEE802.15.4 driver configuration data.

Public Members

struct *ieee802154_config*.[anonymous].[anonymous] `auto_ack_fpb`

`IEEE802154_CONFIG_AUTO_ACK_FPB`

struct *ieee802154_config*.[anonymous].[anonymous] `ack_fpb`

`IEEE802154_CONFIG_ACK_FPB`

bool `pan_coordinator`

`IEEE802154_CONFIG_PAN_COORDINATOR`

bool `promiscuous`

`IEEE802154_CONFIG_PROMISCUOUS`

ieee802154_event_cb_t `event_handler`

`IEEE802154_CONFIG_EVENT_HANDLER`

struct *ieee802154_key* *`mac_keys`

`IEEE802154_CONFIG_MAC_KEYS` Pointer to an array containing a list of keys used for MAC encryption. Refer to secKeyIdLookupDescriptor and secKeyDescriptor in IEEE 802.15.4

key_value field points to a buffer containing the 16 byte key. The buffer is copied by the callee.

The variable length array is terminated by key_value field set to NULL.

uint32_t `frame_counter`

`IEEE802154_CONFIG_FRAME_COUNTER`

struct *ieee802154_config*.[anonymous].[anonymous] `rx_slot`

`IEEE802154_CONFIG_RX_SLOT`

uint32_t `csl_period`

`IEEE802154_CONFIG_CSL_PERIOD`

uint32_t `csl_rx_time`

`IEEE802154_CONFIG_CSL_RX_TIME`

const uint8_t *`ext_addr`

The extended address is expected to be passed starting with the leftmost octet and ending with the rightmost octet. A device with an extended address 01:23:45:67:89:ab:cd:ef should provide a pointer to array containing values in the same exact order.

struct *ieee802154_config*.[anonymous].[anonymous] `ack_ie`

`IEEE802154_CONFIG_ENH_ACK_HEADER_IE`

union *ieee802154_config*.[anonymous] `[anonymous]`

Configuration data.

struct `ieee802154_radio_api`

#include <ieee802154_radio.h> IEEE 802.15.4 radio interface API.

Public Members

struct net_if_api `iface_api`

Mandatory to get in first position. A network device should indeed provide a pointer on such net_if_api structure. So we make current structure pointer that can be casted to a net_if_api structure pointer.

enum *ieee802154_hw_caps* (*`get_capabilities`)(const struct *device* *dev)

Get the device capabilities

int (*`cca`)(const struct *device* *dev)

Clear Channel Assesment - Check channel's activity

int (*`set_channel`)(const struct *device* *dev, uint16_t channel)

Set current channel

int (*`filter`)(const struct *device* *dev, bool set, enum *ieee802154_filter_type* type, const struct *ieee802154_filter* *filter)

Set/Unset filters (for IEEE802154_HW_FILTER)

int (*`set_txpower`)(const struct *device* *dev, int16_t dbm)

Set TX power level in dbm

int (*`tx`)(const struct *device* *dev, enum *ieee802154_tx_mode* mode, struct *net_pkt* *pkt, struct *net_buf* *frag)

Transmit a packet fragment

int (*`start`)(const struct *device* *dev)

Start the device

int (*`stop`)(const struct *device* *dev)

Stop the device

int (*`configure`)(const struct *device* *dev, enum *ieee802154_config_type* type, const struct *ieee802154_config* *config)

Set specific radio driver configuration.

uint16_t (*`get_subg_channel_count`)(const struct *device* *dev)

Get the available amount of Sub-GHz channels

int (*`ed_scan`)(const struct *device* *dev, uint16_t duration, *energy_scan_done_cb_t* done_cb)

Run an energy detection scan. Note: channel must be set prior to request this function. duration parameter is in ms.

uint64_t (*`get_time`)(const struct *device* *dev)

Get the current radio time in microseconds

uint8_t (*`get_sch_acc`)(const struct *device* *dev)

Get the current accuracy, in units of ± ppm, of the clock used for scheduling delayed receive or transmit radio operations. Note: Implementations may optimize this value based on operational conditions (i.e.: temperature).

IEEE 802.15.4 Management

group ieee802154_mgmt

IEEE 802.15.4 library.

Defines

NET_REQUEST_IEEE802154_SET_ACK

NET_REQUEST_IEEE802154_UNSET_ACK

NET_REQUEST_IEEE802154_PASSIVE_SCAN

NET_REQUEST_IEEE802154_ACTIVE_SCAN

NET_REQUEST_IEEE802154_CANCEL_SCAN

NET_REQUEST_IEEE802154_ASSOCIATE

NET_REQUEST_IEEE802154_DISASSOCIATE

NET_REQUEST_IEEE802154_SET_CHANNEL

NET_REQUEST_IEEE802154_GET_CHANNEL

NET_REQUEST_IEEE802154_SET_PAN_ID

NET_REQUEST_IEEE802154_GET_PAN_ID

NET_REQUEST_IEEE802154_SET_EXT_ADDR

NET_REQUEST_IEEE802154_GET_EXT_ADDR

NET_REQUEST_IEEE802154_SET_SHORT_ADDR

NET_REQUEST_IEEE802154_GET_SHORT_ADDR

NET_REQUEST_IEEE802154_GET_TX_POWER

NET_REQUEST_IEEE802154_SET_TX_POWER

NET_EVENT_IEEE802154_SCAN_RESULT

IEEE802154_IS_CHAN_SCANNED(_channel_set, _chan)

IEEE802154_IS_CHAN_UNSCANNED(_channel_set, _chan)

IEEE802154_ALL_CHANNELS

Enums

enum net_request_ieee802154_cmd

Values:

enumerator NET_REQUEST_IEEE802154_CMD_SET_ACK = 1

enumerator NET_REQUEST_IEEE802154_CMD_UNSET_ACK

enumerator NET_REQUEST_IEEE802154_CMD_PASSIVE_SCAN

enumerator NET_REQUEST_IEEE802154_CMD_ACTIVE_SCAN

enumerator NET_REQUEST_IEEE802154_CMD_CANCEL_SCAN

enumerator NET_REQUEST_IEEE802154_CMD_ASSOCIATE

enumerator NET_REQUEST_IEEE802154_CMD_DISASSOCIATE

enumerator NET_REQUEST_IEEE802154_CMD_SET_CHANNEL

enumerator NET_REQUEST_IEEE802154_CMD_GET_CHANNEL

enumerator NET_REQUEST_IEEE802154_CMD_SET_PAN_ID

enumerator NET_REQUEST_IEEE802154_CMD_GET_PAN_ID

enumerator NET_REQUEST_IEEE802154_CMD_SET_EXT_ADDR

enumerator NET_REQUEST_IEEE802154_CMD_GET_EXT_ADDR

enumerator NET_REQUEST_IEEE802154_CMD_SET_SHORT_ADDR

enumerator NET_REQUEST_IEEE802154_CMD_GET_SHORT_ADDR

enumerator NET_REQUEST_IEEE802154_CMD_GET_TX_POWER

enumerator NET_REQUEST_IEEE802154_CMD_SET_TX_POWER

enumerator NET_REQUEST_IEEE802154_CMD_SET_SECURITY_SETTINGS

enumerator NET_REQUEST_IEEE802154_CMD_GET_SECURITY_SETTINGS

enum net_event_ieee802154_cmd

Values:

enumerator NET_EVENT_IEEE802154_CMD_SCAN_RESULT = 1

struct ieee802154_req_params

#include <ieee802154_mgmt.h> Scanning parameters.

Used to request a scan and get results as well

Public Members

uint32_t channel_set

The set of channels to scan, use above macros to manage it

uint32_t duration

Duration of scan, per-channel, in milliseconds

uint16_t channel

Current channel in use as a result

uint16_t pan_id

Current pan_id in use as a result

union *ieee802154_req_params*.[anonymous] [anonymous]

Result address

uint8_t len

length of address

uint8_t lqi

Link quality information, between 0 and 255

struct ieee802154_security_params

#include <ieee802154_mgmt.h> Security parameters.

Used to setup the link-layer security settings

Thread protocol

Overview Thread is a low-power mesh networking technology, designed specifically for home automation applications. It is an IPv6-based standard, which uses 6LoWPAN technology over IEEE 802.15.4 protocol. IP connectivity lets you easily connect a Thread mesh network to the internet with a Thread Border Router.

The Thread specification provides a high level of network security. Mesh networks built with Thread are secure - only authenticated devices can join the network and all communications within the mesh are encrypted. More information about Thread protocol can be found at Thread Group website.

Zephyr integrates an open source Thread protocol implementation called OpenThread, documented on the OpenThread website.

Internet connectivity A Thread Border Router is required to connect mesh network to the internet. An open source implementation of Thread Border Router is provided by the OpenThread community. See OpenThread Border Router guide for instructions on how to set up a Border Router.

Sample usage You can try using OpenThread with the Zephyr Echo server and Echo client samples, which provide out-of-the-box configuration for OpenThread. To enable OpenThread support in these samples, build them with `overlay-ot.conf` overlay config file. See sockets-echo-server-sample and sockets-echo-client-sample for details.

Point-to-Point Protocol (PPP) Support

Overview Point-to-Point Protocol (PPP) is a data link layer (layer 2) communications protocol used to establish a direct connection between two nodes. PPP is used over many types of serial links since IP packets cannot be transmitted over a modem line on their own, without some data link protocol.

In Zephyr, each individual PPP link is modelled as a network interface. This is similar to how Linux implements PPP.

PPP support must be enabled at compile time by setting option `CONFIG_NET_PPP` and `CONFIG_NET_L2_PPP`. The PPP support in Zephyr 2.0 is still experimental and the implementation supports only these protocols:

- LCP (Link Control Protocol, RFC1661)
- HDLC (High-level data link control, RFC1662)
- IPCP (IP Control Protocol, RFC1332)
- IPV6CP (IPv6 Control Protocol, RFC5072)

See also the samples/net/sockets/echo_server/overlay-ppp.conf file for configuration option examples. For using PPP with GSM modem, see *Generic GSM Modem* for additional information.

Testing See the net-tools README file for more details on how to test the Zephyr PPP against pppd running in Linux.

7.20.4 Protocols

CoAP

Overview The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks. It provides a convenient API for RESTful Web services that support CoAP's features. For more information about the protocol itself, see IETF RFC7252 The Constrained Application Protocol.

Zephyr provides a CoAP library which supports client and server roles. The library is configurable as per user needs. The Zephyr CoAP library is implemented using plain buffers. Users of the API create sockets for communication and pass the buffer to the library for parsing and other purposes. The library itself doesn't create any sockets for users.

On top of CoAP, Zephyr has support for LWM2M "Lightweight Machine 2 Machine" protocol, a simple, low-cost remote management and service enablement mechanism. See *Lightweight M2M (LWM2M)* for more information.

Supported RFCs:

Supported RFCs:

- RFC7252: The Constrained Application Protocol (CoAP)
- RFC6690: Constrained RESTful Environments (CoRE) Link Format
- RFC7959: Block-Wise Transfers in the Constrained Application Protocol (CoAP)
- RFC7641: Observing Resources in the Constrained Application Protocol (CoAP)

Note: Not all parts of these RFCs are supported. Features are supported based on Zephyr requirements.

Sample Usage

CoAP Server To create a CoAP server, resources for the server need to be defined. The `.well-known/core` resource should be added before all other resources that should be included in the responses of the `.well-known/core` resource.

```
static struct coap_resource resources[] = {
    { .get = well_known_core_get,
      .path = COAP_WELL_KNOWN_CORE_PATH,
    },
    { .get  = sample_get,
      .post = sample_post,
      .del  = sample_del,
      .put  = sample_put,
      .path = sample_path
    },
    { },
};
```

An application reads data from the socket and passes the buffer to the CoAP library to parse the message. If the CoAP message is proper, the library uses the buffer along with resources defined above to call the correct callback function to handle the CoAP request from the client. It's the callback function's responsibility to either reply or act according to CoAP request.

```
coap_packet_parse(&request, data, data_len, options, opt_num);
...
coap_handle_request(&request, resources, options, opt_num,
                    client_addr, client_addr_len);
```

If `CONFIG_COAP_URI_WILDCARD` enabled, server may accept multiple resources using MQTT-like wildcard style:

- the plus symbol represents a single-level wild card in the path;
- the hash symbol represents the multi-level wild card in the path.

```
static const char * const led_set[] = { "led","+","set", NULL };
static const char * const btn_get[] = { "button","#", NULL };
static const char * const no_wc[] = { "test","+1", NULL };
```

It accepts /led/0/set, led/1234/set, led/any/set, /button/door/1, /test/+1, but returns -ENOENT for /led/1, /test/21, /test/1.

This option is enabled by default, disable it to avoid unexpected behaviour with resource path like '/some_resource/+/#'.

CoAP Client If the CoAP client knows about resources in the CoAP server, the client can start prepare CoAP requests and wait for responses. If the client doesn't know about resources in the CoAP server, it can request resources through the `.well-known/core` CoAP message.

```
/* Initialize the CoAP message */
char *path = "test";
struct coap_packet request;
uint8_t data[100];
uint8_t payload[20];

coap_packet_init(&request, data, sizeof(data),
                 1, COAP_TYPE_CON, 8, coap_next_token(),
                 COAP_METHOD_GET, coap_next_id());

/* Append options */
coap_packet_append_option(&request, COAP_OPTION_URI_PATH,
                          path, strlen(path));

/* Append Payload marker if you are going to add payload */
coap_packet_append_payload_marker(&request);

/* Append payload */
coap_packet_append_payload(&request, (uint8_t *)payload,
                           sizeof(payload) - 1);

/* send over sockets */
```

Testing There are various ways to test Zephyr CoAP library.

libcoap libcoap implements a lightweight application-protocol for devices that are resource constrained, such as by computing power, RF range, memory, bandwidth, or network packet sizes. Sources

can be found here libcoap. libcoap has a script (`examples/etsi_coaptest.sh`) to test coap-server functionality in Zephyr.

See the net-tools project for more details

The coap-server-sample sample can be built and executed on QEMU as described in *Networking with QEMU*.

Use this command on the host to run the libcoap implementation of the ETSI test cases:

```
sudo ./libcoap/examples/etsi_coaptest.sh -i tap0 2001:db8::1
```

TTCN3 Eclipse has TTCN3 based tests to run against CoAP implementations.

Install eclipse-titan and set symbolic links for titan tools

```
sudo apt-get install eclipse-titan

cd /usr/share/titan

sudo ln -s /usr/bin bin
sudo ln /usr/bin/titanver bin
sudo ln -s /usr/bin/mctr_cli bin
sudo ln -s /usr/include/titan include
sudo ln -s /usr/lib/titan lib

export TTCN3_DIR=/usr/share/titan

git clone https://github.com/eclipse/titan.misc.git

cd titan.misc
```

Follow the instruction to setup CoAP test suite from here:

- https://github.com/eclipse/titan.misc
- https://github.com/eclipse/titan.misc/tree/master/CoAP_Conf

After the build is complete, the coap-server-sample sample can be built and executed on QEMU as described in *Networking with QEMU*.

Change the client (test suite) and server (Zephyr coap-server sample) addresses in coap.cfg file as per your setup.

Execute the test cases with following command.

```
ttcn3_start coaptests coap.cfg
```

Sample output of ttcn3 tests looks like this.

```
Verdict statistics: 0 none (0.00 %), 10 pass (100.00 %), 0 inconc (0.00 %), 0 fail (0.
↪00 %), 0 error (0.00 %).
Test execution summary: 10 test cases were executed. Overall verdict: pass
```

API Reference

group `coap`

COAP library.

Defines

COAP_REQUEST_MASK

COAP_VERSION_1

coap_make_response_code(class, det)

COAP_CODE_EMPTY

COAP_TOKEN_MAX_LEN

GET_BLOCK_NUM(v)

GET_BLOCK_SIZE(v)

GET_MORE(v)

COAP_DEFAULT_MAX_RETRANSMIT

COAP_DEFAULT_ACK_RANDOM_FACTOR

COAP_WELL_KNOWN_CORE_PATH

This resource should be added before all other resources that should be included in the responses of the .well-known/core resource.

Typedefs

typedef int (*coap_method_t)(struct *coap_resource* *resource, struct *coap_packet* *request, struct *sockaddr* *addr, *socklen_t* addr_len)

Type of the callback being called when a resource's method is invoked by the remote entity.

typedef void (*coap_notify_t)(struct *coap_resource* *resource, struct *coap_observer* *observer)

Type of the callback being called when a resource's has observers to be informed when an update happens.

typedef int (*coap_reply_t)(const struct *coap_packet* *response, struct *coap_reply* *reply, const struct *sockaddr* *from)

Helper function to be called when a response matches the a pending request.

Enums

enum coap_option_num

Set of CoAP packet options we are aware of.

Users may add options other than these to their packets, provided they know how to format them correctly. The only restriction is that all options must be added to a packet in numeric order.

Refer to RFC 7252, section 12.2 for more information.

Values:

enumerator COAP_OPTION_IF_MATCH = 1

enumerator COAP_OPTION_URI_HOST = 3

enumerator COAP_OPTION_ETAG = 4

enumerator COAP_OPTION_IF_NONE_MATCH = 5

enumerator COAP_OPTION_OBSERVE = 6

enumerator COAP_OPTION_URI_PORT = 7

enumerator COAP_OPTION_LOCATION_PATH = 8

enumerator COAP_OPTION_URI_PATH = 11

enumerator COAP_OPTION_CONTENT_FORMAT = 12

enumerator COAP_OPTION_MAX_AGE = 14

enumerator COAP_OPTION_URI_QUERY = 15

enumerator COAP_OPTION_ACCEPT = 17

enumerator COAP_OPTION_LOCATION_QUERY = 20

enumerator COAP_OPTION_BLOCK2 = 23

enumerator COAP_OPTION_BLOCK1 = 27

enumerator COAP_OPTION_SIZE2 = 28

enumerator COAP_OPTION_PROXY_URI = 35

enumerator COAP_OPTION_PROXY_SCHEME = 39

enumerator COAP_OPTION_SIZE1 = 60

enum coap_method

Available request methods.

To be used when creating a request or a response.

Values:

enumerator COAP_METHOD_GET = 1

enumerator COAP_METHOD_POST = 2

enumerator COAP_METHOD_PUT = 3

enumerator COAP_METHOD_DELETE = 4

enum coap_msgtype

CoAP packets may be of one of these types.

Values:

enumerator COAP_TYPE_CON = 0

Confirmable message.

The packet is a request or response the destination end-point must acknowledge.

enumerator COAP_TYPE_NON_CON = 1

Non-confirmable message.

The packet is a request or response that doesn't require acknowledgements.

enumerator COAP_TYPE_ACK = 2

Acknowledge.

Response to a confirmable message.

enumerator COAP_TYPE_RESET = 3

Reset.

Rejecting a packet for any reason is done by sending a message of this type.

enum coap_response_code

Set of response codes available for a response packet.

To be used when creating a response.

Values:

enumerator COAP_RESPONSE_CODE_OK = ((2 << 5) | (0))

enumerator COAP_RESPONSE_CODE_CREATED = ((2 << 5) | (1))

enumerator COAP_RESPONSE_CODE_DELETED = ((2 << 5) | (2))

enumerator COAP_RESPONSE_CODE_VALID = ((2 << 5) | (3))

enumerator COAP_RESPONSE_CODE_CHANGED = ((2 << 5) | (4))

enumerator COAP_RESPONSE_CODE_CONTENT = ((2 << 5) | (5))

enumerator COAP_RESPONSE_CODE_CONTINUE = ((2 << 5) | (31))

enumerator COAP_RESPONSE_CODE_BAD_REQUEST = ((4 << 5) | (0))

enumerator COAP_RESPONSE_CODE_UNAUTHORIZED = ((4 << 5) | (1))

enumerator COAP_RESPONSE_CODE_BAD_OPTION = ((4 << 5) | (2))

enumerator COAP_RESPONSE_CODE_FORBIDDEN = ((4 << 5) | (3))

enumerator COAP_RESPONSE_CODE_NOT_FOUND = ((4 << 5) | (4))

enumerator COAP_RESPONSE_CODE_NOT_ALLOWED = ((4 << 5) | (5))

enumerator COAP_RESPONSE_CODE_NOT_ACCEPTABLE = ((4 << 5) | (6))

enumerator COAP_RESPONSE_CODE_INCOMPLETE = ((4 << 5) | (8))

enumerator COAP_RESPONSE_CODE_PRECONDITION_FAILED = ((4 << 5) | (12))

enumerator COAP_RESPONSE_CODE_REQUEST_TOO_LARGE = ((4 << 5) | (13))

enumerator COAP_RESPONSE_CODE_UNSUPPORTED_CONTENT_FORMAT = ((4 << 5) | (15))

enumerator COAP_RESPONSE_CODE_INTERNAL_ERROR = ((5 << 5) | (0))

enumerator COAP_RESPONSE_CODE_NOT_IMPLEMENTED = ((5 << 5) | (1))

enumerator COAP_RESPONSE_CODE_BAD_GATEWAY = ((5 << 5) | (2))

enumerator COAP_RESPONSE_CODE_SERVICE_UNAVAILABLE = ((5 << 5) | (3))

enumerator COAP_RESPONSE_CODE_GATEWAY_TIMEOUT = ((5 << 5) | (4))

enumerator COAP_RESPONSE_CODE_PROXYING_NOT_SUPPORTED = ((5 << 5) | (5))

enum coap_content_format

Set of Content-Format option values for CoAP.

To be used when encoding or decoding a Content-Format option.

Values:

enumerator COAP_CONTENT_FORMAT_TEXT_PLAIN = 0

enumerator COAP_CONTENT_FORMAT_APP_LINK_FORMAT = 40

enumerator COAP_CONTENT_FORMAT_APP_XML = 41

enumerator COAP_CONTENT_FORMAT_APP_OCTET_STREAM = 42

enumerator COAP_CONTENT_FORMAT_APP_EXI = 47

enumerator COAP_CONTENT_FORMAT_APP_JSON = 50

enumerator COAP_CONTENT_FORMAT_APP_CBOR = 60

enum coap_block_size

Represents the size of each block that will be transferred using block-wise transfers [RFC7959]:

Each entry maps directly to the value that is used in the wire.

https://tools.ietf.org/html/rfc7959

Values:

enumerator COAP_BLOCK_16

enumerator COAP_BLOCK_32

enumerator COAP_BLOCK_64

enumerator COAP_BLOCK_128

enumerator COAP_BLOCK_256

enumerator COAP_BLOCK_512

enumerator COAP_BLOCK_1024

Functions

uint8_t coap_header_get_version(const struct *coap_packet* *cpkt)

Returns the version present in a CoAP packet.

Parameters

- cpkt – CoAP packet representation

Returns the CoAP version in packet

uint8_t coap_header_get_type(const struct *coap_packet* *cpkt)

Returns the type of the CoAP packet.

Parameters

- cpkt – CoAP packet representation

Returns the type of the packet

uint8_t coap_header_get_token(const struct *coap_packet* *cpkt, uint8_t *token)

Returns the token (if any) in the CoAP packet.

Parameters

- cpkt – CoAP packet representation
- token – Where to store the token, must point to a buffer containing at least COAP_TOKEN_MAX_LEN bytes

Returns Token length in the CoAP packet (0 - COAP_TOKEN_MAX_LEN).

uint8_t coap_header_get_code(const struct *coap_packet* *cpkt)

Returns the code of the CoAP packet.

Parameters

- cpkt – CoAP packet representation

Returns the code present in the packet

uint16_t coap_header_get_id(const struct *coap_packet* *cpkt)

Returns the message id associated with the CoAP packet.

Parameters

- cpkt – CoAP packet representation

Returns the message id present in the packet

const uint8_t *coap_packet_get_payload(const struct *coap_packet* *cpkt, uint16_t *len)

Returns the data pointer and length of the CoAP packet.

Parameters

- cpkt – CoAP packet representation
- len – Total length of CoAP payload

Returns data pointer and length if payload exists NULL pointer and length set to 0 in case there is no payload

int coap_packet_parse(struct *coap_packet* *cpkt, uint8_t *data, uint16_t len, struct *coap_option* *options, uint8_t opt_num)

Parses the CoAP packet in data, validating it and initializing *cpkt*. *data* must remain valid while *cpkt* is used.

Parameters

- cpkt – Packet to be initialized from received *data*.
- data – Data containing a CoAP packet, its *data* pointer is positioned on the start of the CoAP packet.
- len – Length of the data
- options – Parse options and cache its details.
- opt_num – Number of options

Returns 0 in case of success or negative in case of error.

int coap_packet_init(struct *coap_packet* *cpkt, uint8_t *data, uint16_t max_len, uint8_t ver, uint8_t type, uint8_t token_len, const uint8_t *token, uint8_t code, uint16_t id)

Creates a new CoAP Packet from input data.

Parameters

- cpkt – New packet to be initialized using the storage from *data*.

- `data` – Data that will contain a CoAP packet information
- `max_len` – Maximum allowable length of data
- `ver` – CoAP header version
- `type` – CoAP header type
- `token_len` – CoAP header token length
- `token` – CoAP header token
- `code` – CoAP header code
- `id` – CoAP header message id

Returns 0 in case of success or negative in case of error.

int `coap_ack_init`(struct *coap_packet* *cpkt, const struct *coap_packet* *req, uint8_t *data, uint16_t max_len, uint8_t code)

Create a new CoAP Acknowledgment message for given request.

This function works like *coap_packet_init*, filling CoAP header type, CoAP header token, and CoAP header message id fields according to acknowledgment rules.

Parameters

- `cpkt` – New packet to be initialized using the storage from *data*.
- `req` – CoAP request packet that is being acknowledged
- `data` – Data that will contain a CoAP packet information
- `max_len` – Maximum allowable length of data
- `code` – CoAP header code

Returns 0 in case of success or negative in case of error.

uint8_t *`coap_next_token`(void)

Returns a randomly generated array of 8 bytes, that can be used as a message's token.

Returns a 8-byte pseudo-random token.

uint16_t `coap_next_id`(void)

Helper to generate message ids.

Returns a new message id

int `coap_find_options`(const struct *coap_packet* *cpkt, uint16_t code, struct *coap_option* *options, uint16_t veclen)

Return the values associated with the option of value *code*.

Parameters

- `cpkt` – CoAP packet representation
- `code` – Option number to look for
- `options` – Array of *coap_option* where to store the value of the options found
- `veclen` – Number of elements in the options array

Returns The number of options found in packet matching code, negative on error.

int `coap_packet_append_option`(struct *coap_packet* *cpkt, uint16_t code, const uint8_t *value, uint16_t len)

Appends an option to the packet.

Note: options must be added in numeric order of their codes. Otherwise error will be returned. TODO: Add support for placing options according to its delta value.

Parameters

- `cpkt` – Packet to be updated
- `code` – Option code to add to the packet, see *coap_option_num*
- `value` – Pointer to the value of the option, will be copied to the packet
- `len` – Size of the data to be added

Returns 0 in case of success or negative in case of error.

unsigned int `coap_option_value_to_int`(const struct *coap_option* *option)

Converts an option to its integer representation.

Assumes that the number is encoded in the network byte order in the option.

Parameters

- `option` – Pointer to the option value, retrieved by *coap_find_options()*

Returns The integer representation of the option

int `coap_append_option_int`(struct *coap_packet* *cpkt, uint16_t code, unsigned int val)

Appends an integer value option to the packet.

The option must be added in numeric order of their codes, and the least amount of bytes will be used to encode the value.

Parameters

- `cpkt` – Packet to be updated
- `code` – Option code to add to the packet, see *coap_option_num*
- `val` – Integer value to be added

Returns 0 in case of success or negative in case of error.

int `coap_packet_append_payload_marker`(struct *coap_packet* *cpkt)

Append payload marker to CoAP packet.

Parameters

- `cpkt` – Packet to append the payload marker (0xFF)

Returns 0 in case of success or negative in case of error.

int `coap_packet_append_payload`(struct *coap_packet* *cpkt, const uint8_t *payload, uint16_t payload_len)

Append payload to CoAP packet.

Parameters

- `cpkt` – Packet to append the payload
- `payload` – CoAP packet payload
- `payload_len` – CoAP packet payload len

Returns 0 in case of success or negative in case of error.

int `coap_handle_request`(struct *coap_packet* *cpkt, struct *coap_resource* *resources, struct *coap_option* *options, uint8_t opt_num, struct *sockaddr* *addr, *socklen_t* addr_len)

When a request is received, call the appropriate methods of the matching resources.

Parameters

- `cpkt` – Packet received
- `resources` – Array of known resources

- `options` – Parsed options from *coap_packet_parse()*
- `opt_num` – Number of options
- `addr` – Peer address
- `addr_len` – Peer address length

Returns 0 in case of success or negative in case of error.

static inline uint16_t `coap_block_size_to_bytes`(enum *coap_block_size* block_size)

Helper for converting the enumeration to the size expressed in bytes.

Parameters

- `block_size` – The block size to be converted

Returns The size in bytes that the block_size represents

int `coap_block_transfer_init`(struct *coap_block_context* *ctx, enum *coap_block_size* block_size, size_t total_size)

Initializes the context of a block-wise transfer.

Parameters

- `ctx` – The context to be initialized
- `block_size` – The size of the block
- `total_size` – The total size of the transfer, if known

Returns 0 in case of success or negative in case of error.

int `coap_append_block1_option`(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append BLOCK1 option to the packet.

Parameters

- `cpkt` – Packet to be updated
- `ctx` – Block context from which to retrieve the information for the Block1 option

Returns 0 in case of success or negative in case of error.

int `coap_append_block2_option`(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append BLOCK2 option to the packet.

Parameters

- `cpkt` – Packet to be updated
- `ctx` – Block context from which to retrieve the information for the Block2 option

Returns 0 in case of success or negative in case of error.

int `coap_append_size1_option`(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append SIZE1 option to the packet.

Parameters

- `cpkt` – Packet to be updated
- `ctx` – Block context from which to retrieve the information for the Size1 option

Returns 0 in case of success or negative in case of error.

int `coap_append_size2_option`(struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Append SIZE2 option to the packet.

Parameters

- `cpkt` – Packet to be updated
- `ctx` – Block context from which to retrieve the information for the Size2 option

Returns 0 in case of success or negative in case of error.

int `coap_get_option_int`(const struct *coap_packet* *cpkt, uint16_t code)

Get the integer representation of a CoAP option.

Parameters

- `cpkt` – Packet to be inspected
- `code` – CoAP option code

Returns Integer value >= 0 in case of success or negative in case of error.

int `coap_update_from_block`(const struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Retrieves BLOCK{1,2} and SIZE{1,2} from *cpkt* and updates *ctx* accordingly.

Parameters

- `cpkt` – Packet in which to look for block-wise transfers options
- `ctx` – Block context to be updated

Returns 0 in case of success or negative in case of error.

size_t `coap_next_block`(const struct *coap_packet* *cpkt, struct *coap_block_context* *ctx)

Updates *ctx* so after this is called the current entry indicates the correct offset in the body of data being transferred.

Parameters

- `cpkt` – Packet in which to look for block-wise transfers options
- `ctx` – Block context to be updated

Returns The offset in the block-wise transfer, 0 if the transfer has finished.

void `coap_observer_init`(struct *coap_observer* *observer, const struct *coap_packet* *request, const struct *sockaddr* *addr)

Indicates that the remote device referenced by *addr*, with *request*, wants to observe a resource.

Parameters

- `observer` – Observer to be initialized
- `request` – Request on which the observer will be based
- `addr` – Address of the remote device

bool `coap_register_observer`(struct *coap_resource* *resource, struct *coap_observer* *observer)

After the observer is initialized, associate the observer with an resource.

Parameters

- `resource` – Resource to add an observer
- `observer` – Observer to be added

Returns true if this is the first observer added to this resource.

void coap_remove_observer(struct *coap_resource* *resource, struct *coap_observer* *observer)

Remove this observer from the list of registered observers of that resource.

Parameters

- resource – Resource in which to remove the observer
- observer – Observer to be removed

struct *coap_observer* *coap_find_observer_by_addr(struct *coap_observer* *observers, size_t len, const struct *sockaddr* *addr)

Returns the observer that matches address *addr*.

Parameters

- observers – Pointer to the array of observers
- len – Size of the array of observers
- addr – Address of the endpoint observing a resource

Returns A pointer to a observer if a match is found, NULL otherwise.

struct *coap_observer* *coap_observer_next_unused(struct *coap_observer* *observers, size_t len)

Returns the next available observer representation.

Parameters

- observers – Pointer to the array of observers
- len – Size of the array of observers

Returns A pointer to a observer if there's an available observer, NULL otherwise.

void coap_reply_init(struct *coap_reply* *reply, const struct *coap_packet* *request)

Indicates that a reply is expected for *request*.

Parameters

- reply – Reply structure to be initialized
- request – Request from which *reply* will be based

int coap_pending_init(struct *coap_pending* *pending, const struct *coap_packet* *request, const struct *sockaddr* *addr, uint8_t retries)

Initialize a pending request with a request.

The request's fields are copied into the pending struct, so *request* doesn't have to live for as long as the pending struct lives, but "data" that needs to live for at least that long.

Parameters

- pending – Structure representing the waiting for a confirmation message, initialized with data from *request*
- request – Message waiting for confirmation
- addr – Address to send the retransmission
- retries – Maximum number of retransmissions of the message.

Returns 0 in case of success or negative in case of error.

struct *coap_pending* *coap_pending_next_unused(struct *coap_pending* *pendings, size_t len)

Returns the next available pending struct, that can be used to track the retransmission status of a request.

Parameters

- pendings – Pointer to the array of *coap_pending* structures
- len – Size of the array of *coap_pending* structures

Returns pointer to a free *coap_pending* structure, NULL in case none could be found.

struct *coap_reply* *coap_reply_next_unused(struct *coap_reply* *replies, size_t len)

Returns the next available reply struct, so it can be used to track replies and notifications received.

Parameters

- `replies` – Pointer to the array of *coap_reply* structures
- `len` – Size of the array of *coap_reply* structures

Returns pointer to a free *coap_reply* structure, NULL in case none could be found.

struct *coap_pending* *coap_pending_received(const struct *coap_packet* *response, struct *coap_pending* *pendings, size_t len)

After a response is received, returns if there is any matching pending request exits. User has to clear all pending retransmissions related to that response by calling *coap_pending_clear()*.

Parameters

- `response` – The received response
- `pendings` – Pointer to the array of *coap_reply* structures
- `len` – Size of the array of *coap_reply* structures

Returns pointer to the associated *coap_pending* structure, NULL in case none could be found.

struct *coap_reply* *coap_response_received(const struct *coap_packet* *response, const struct *sockaddr* *from, struct *coap_reply* *replies, size_t len)

After a response is received, call coap_reply_t handler registered in *coap_reply* structure.

Parameters

- `response` – A response received
- `from` – Address from which the response was received
- `replies` – Pointer to the array of *coap_reply* structures
- `len` – Size of the array of *coap_reply* structures

Returns Pointer to the reply matching the packet received, NULL if none could be found.

struct *coap_pending* *coap_pending_next_to_expire(struct *coap_pending* *pendings, size_t len)

Returns the next pending about to expire, pending->timeout informs how many ms to next expiration.

Parameters

- `pendings` – Pointer to the array of *coap_pending* structures
- `len` – Size of the array of *coap_pending* structures

Returns The next *coap_pending* to expire, NULL if none is about to expire.

bool coap_pending_cycle(struct *coap_pending* *pending)

After a request is sent, user may want to cycle the pending retransmission so the timeout is updated.

Parameters

- `pending` – Pending representation to have its timeout updated

Returns false if this is the last retransmission.

void coap_pending_clear(struct *coap_pending* *pending)

Cancels the pending retransmission, so it again becomes available.

Parameters

- pending – Pending representation to be canceled

void coap_pendings_clear(struct *coap_pending* *pendings, size_t len)

Cancels all pending retransmissions, so they become available again.

Parameters

- pendings – Pointer to the array of *coap_pending* structures
- len – Size of the array of *coap_pending* structures

void coap_reply_clear(struct *coap_reply* *reply)

Cancels awaiting for this reply, so it becomes available again. User responsibility to free the memory associated with data.

Parameters

- reply – The reply to be canceled

void coap_replies_clear(struct *coap_reply* *replies, size_t len)

Cancels all replies, so they become available again.

Parameters

- replies – Pointer to the array of *coap_reply* structures
- len – Size of the array of *coap_reply* structures

int coap_resource_notify(struct *coap_resource* *resource)

Indicates that this resource was updated and that the *notify* callback should be called for every registered observer.

Parameters

- resource – Resource that was updated

Returns 0 in case of success or negative in case of error.

bool coap_request_is_observe(const struct *coap_packet* *request)

Returns if this request is enabling observing a resource.

Parameters

- request – Request to be checked

Returns True if the request is enabling observing a resource, False otherwise

int coap_well_known_core_get(struct *coap_resource* *resource, struct *coap_packet* *request, struct *coap_packet* *response, uint8_t *data, uint16_t len)

struct coap_resource

#include <coap.h> Description of CoAP resource.

CoAP servers often want to register resources, so that clients can act on them, by fetching their state or requesting updates to them.

Public Members

coap_method_t get

Which function to be called for each CoAP method

struct `coap_observer`

#include <coap.h> Represents a remote device that is observing a local resource.

struct `coap_packet`

#include <coap.h> Representation of a CoAP Packet.

struct `coap_option`

#include <coap.h>

struct `coap_pending`

#include <coap.h> Represents a request awaiting for an acknowledgment (ACK).

struct `coap_reply`

#include <coap.h> Represents the handler for the reply of a request, it is also used when observing resources.

struct `coap_block_context`

#include <coap.h> Represents the current state of a block-wise transaction.

struct `coap_core_metadata`

#include <coap_link_format.h> In case you want to add attributes to the resources included in the 'well-known/core' "virtual" resource, the 'user_data' field should point to a valid *coap_core_metadata* structure.

Lightweight M2M (LWM2M)

Overview Lightweight Machine to Machine (LwM2M) is an application layer protocol designed with device management, data reporting and device actuation in mind. Based on CoAP/UDP, LwM2M is a standard defined by the Open Mobile Alliance and suitable for constrained devices by its use of CoAP packet-size optimization and a simple, stateless flow that supports a REST API.

One of the key differences between LwM2M and CoAP is that an LwM2M client initiates the connection to an LwM2M server. The server can then use the REST API to manage various interfaces with the client.

LwM2M uses a simple resource model with the core set of objects and resources defined in the specification.

Example LwM2M object and resources: Device *Object definition*

Object ID	Name	Instance	Mandatory
3	Device	Single	Mandatory

Resource definitions

`* R=Read, W=Write, E=Execute`

ID	Name	OP*	Instance	Mandatory	Type
0	Manufacturer	R	Single	Optional	String
1	Model	R	Single	Optional	String
2	Serial number	R	Single	Optional	String
3	Firmware version	R	Single	Optional	String
4	Reboot	E	Single	Mandatory	
5	Factory Reset	E	Single	Optional	
6	Available Power Sources	R	Multiple	Optional	Integer 0-7
7	Power Source Voltage (mV)	R	Multiple	Optional	Integer
8	Power Source Current (mA)	R	Multiple	Optional	Integer
9	Battery Level %	R	Single	Optional	Integer
10	Memory Free (Kb)	R	Single	Optional	Integer
11	Error Code	R	Multiple	Optional	Integer 0-8
12	Reset Error	E	Single	Optional	
13	Current Time	RW	Single	Optional	Time
14	UTC Offset	RW	Single	Optional	String
15	Timezone	RW	Single	Optional	String
16	Supported Binding	R	Single	Mandatory	String
17	Device Type	R	Single	Optional	String
18	Hardware Version	R	Single	Optional	String
19	Software Version	R	Single	Optional	String
20	Battery Status	R	Single	Optional	Integer 0-6
21	Memory Total (Kb)	R	Single	Optional	Integer
22	ExtDevInfo	R	Multiple	Optional	ObjLnk

The server could query the `Manufacturer` resource for `Device` object instance 0 (the default and only instance) by sending a `READ 3/0/0` operation to the client.

The full list of registered objects and resource IDs can be found in the LwM2M registry.

Zephyr's LwM2M library lives in the subsys/net/lib/lwm2m, with a client sample in samples/net/lwm2m_client. For more information about the provided sample see: lwm2m-client-sample The sample can be configured to use normal unsecure network sockets or sockets secured via DTLS.

The Zephyr LwM2M library implements the following items:

- engine to process networking events and core functions
- RD client which performs BOOTSTRAP and REGISTRATION functions
- TLV, JSON, and plain text formatting functions
- LwM2M Technical Specification Enabler objects such as Security, Server, Device, Firmware Update, etc.
- Extended IPSO objects such as Light Control, Temperature Sensor, and Timer

The library currently implements up to LwM2M specification 1.0.2.

For more information about LwM2M visit OMA Specworks LwM2M.

Sample usage To use the LwM2M library, start by creating an LwM2M client context *lwm2m_ctx* structure:

```
/* LwM2M client context */
static struct lwm2m_ctx client;
```

Create callback functions for LwM2M resource exuctions:

```
static int device_reboot_cb(uint16_t obj_inst_id, uint8_t *args,
                            uint16_t args_len)
{
        LOG_INF("Device rebooting.");
        LOG_PANIC();
        sys_reboot(0);
        return 0; /* wont reach this */
}
```

The LwM2M RD client can send events back to the sample. To receive those events, setup a callback function:

```
static void rd_client_event(struct lwm2m_ctx *client,
                            enum lwm2m_rd_client_event client_event)
{
        switch (client_event) {

        case LWM2M_RD_CLIENT_EVENT_NONE:
                /* do nothing */
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_FAILURE:
                LOG_DBG("Bootstrap registration failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_COMPLETE:
                LOG_DBG("Bootstrap registration complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_TRANSFER_COMPLETE:
                LOG_DBG("Bootstrap transfer complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_REGISTRATION_FAILURE:
                LOG_DBG("Registration failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE:
                LOG_DBG("Registration complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_UPDATE_FAILURE:
                LOG_DBG("Registration update failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE:
                LOG_DBG("Registration update complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_DEREGISTER_FAILURE:
                LOG_DBG("Deregister failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_DISCONNECT:
                LOG_DBG("Disconnected");
                break;
```

(continues on next page)

(continued from previous page)

```
        }
}
```

Next we assign `Security` resource values to let the client know where and how to connect as well as set the `Manufacturer` and `Reboot` resources in the `Device` object with some data and the callback we defined above:

```
/*
 * Server URL of default Security object = 0/0/0
 * Use leshan.eclipse.org server IP (5.39.83.206) for connection
 */
lwm2m_engine_set_string("0/0/0", "coap://5.39.83.206");

/*
 * Security Mode of default Security object = 0/0/2
 * 3 = NoSec mode (no security beware!)
 */
lwm2m_engine_set_u8("0/0/2", 3);

#define CLIENT_MANUFACTURER "Zephyr Manufacturer"

/*
 * Manufacturer resource of Device object = 3/0/0
 * We use lwm2m_engine_set_res_data() function to set a pointer to the
 * CLIENT_MANUFACTURER string.
 * Note the LWM2M_RES_DATA_FLAG_RO flag which stops the engine from
 * trying to assign a new value to the buffer.
 */
lwm2m_engine_set_res_data("3/0/0", CLIENT_MANUFACTURER,
                          sizeof(CLIENT_MANUFACTURER),
                          LWM2M_RES_DATA_FLAG_RO);

/* Reboot resource of Device object = 3/0/4 */
lwm2m_engine_register_exec_callback("3/0/4", device_reboot_cb);
```

Lastly, we start the LwM2M RD client (which in turn starts the LwM2M engine). The second parameter of *`lwm2m_rd_client_start()`* is the client endpoint name. This is important as it needs to be unique per LwM2M server:

```
(void)memset(&client, 0x0, sizeof(client));
lwm2m_rd_client_start(&client, "unique-endpoint-name", 0, rd_client_event);
```

Using LwM2M library with DTLS The Zephyr LwM2M library can be used with DTLS transport for secure communication by selecting `CONFIG_LWM2M_DTLS_SUPPORT`. In the client initialization we need to create a PSK and identity. These need to match the security information loaded onto the LwM2M server. Normally, the endpoint name is used to lookup the related security information:

```
/* "000102030405060708090a0b0c0d0e0f" */
static unsigned char client_psk[] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
};

static const char client_identity[] = "Client_identity";
```

Next we alter the `Security` object resources to include DTLS security information. The server URL

should begin with `coaps://` to indicate security is required. Assign a 0 value (Pre-shared Key mode) to the `Security Mode` resource. Lastly, set the client identity and PSK resources.

```
/* Use coaps:// for server URL protocol */
lwm2m_engine_set_string("0/0/0", "coaps://5.39.83.206");
/* 0 = Pre-Shared Key mode */
lwm2m_engine_set_u8("0/0/2", 0);
/* Set the client identity */
lwm2m_engine_set_string("0/0/3", (char *)client_identity);
/* Set the client pre-shared key (PSK) */
lwm2m_engine_set_opaque("0/0/5", (void *)client_psk, sizeof(client_psk));
```

Before calling *lwm2m_rd_client_start()* assign the tls_tag # where the LwM2M library should store the DTLS information prior to connection (normally a value of 1 is ok here).

```
(void)memset(&client, 0x0, sizeof(client));
client.tls_tag = 1; /* <---- */
lwm2m_rd_client_start(&client, "endpoint-name", 0, rd_client_event);
```

For a more detailed LwM2M client sample see: lwm2m-client-sample.

API Reference

group `lwm2m_api`

LwM2M high-level API.

LwM2M high-level interface is defined in this header.

Note: The implementation assumes UDP module is enabled.

Note: LwM2M 1.0.x is currently the only supported version.

Defines

`LWM2M_OBJECT_SECURITY_ID`

LwM2M Objects managed by OMA for LwM2M tech specification. Objects in this range have IDs from 0 to 1023. For more information refer to Technical Specification OMA-TS-LightweightM2M-V1_0_2-20180209-A.

`LWM2M_OBJECT_SERVER_ID`

`LWM2M_OBJECT_ACCESS_CONTROL_ID`

`LWM2M_OBJECT_DEVICE_ID`

`LWM2M_OBJECT_CONNECTIVITY_MONITORING_ID`

`LWM2M_OBJECT_FIRMWARE_ID`

`LWM2M_OBJECT_LOCATION_ID`

LWM2M_OBJECT_CONNECTIVITY_STATISTICS_ID

IPSO_OBJECT_GENERIC_SENSOR_ID

LwM2M Objects produced by 3rd party Standards Development Organizations. Objects in this range have IDs from 2048 to 10240 Refer to the OMA LightweightM2M (LwM2M) Object and Resource Registry: http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html.

IPSO_OBJECT_TEMP_SENSOR_ID

IPSO_OBJECT_HUMIDITY_SENSOR_ID

IPSO_OBJECT_LIGHT_CONTROL_ID

IPSO_OBJECT_ACCELEROMETER_ID

IPSO_OBJECT_PRESSURE_ID

IPSO_OBJECT_BUZZER_ID

IPSO_OBJECT_TIMER_ID

IPSO_OBJECT_ONOFF_SWITCH_ID

IPSO_OBJECT_PUSH_BUTTON_ID

LWM2M_DEVICE_PWR_SRC_TYPE_DC_POWER

Power source types used for the "Available Power Sources" resource of the LwM2M Device object.

LWM2M_DEVICE_PWR_SRC_TYPE_BAT_INT

LWM2M_DEVICE_PWR_SRC_TYPE_BAT_EXT

LWM2M_DEVICE_PWR_SRC_TYPE_UNUSED

LWM2M_DEVICE_PWR_SRC_TYPE_PWR_OVER_ETH

LWM2M_DEVICE_PWR_SRC_TYPE_USB

LWM2M_DEVICE_PWR_SRC_TYPE_AC_POWER

LWM2M_DEVICE_PWR_SRC_TYPE_SOLAR

LWM2M_DEVICE_PWR_SRC_TYPE_MAX

LWM2M_DEVICE_ERROR_NONE

Error codes used for the “Error Code” resource of the LwM2M Device object. An LwM2M client can register one of the following error codes via the *lwm2m_device_add_err()* function.

LWM2M_DEVICE_ERROR_LOW_POWER

LWM2M_DEVICE_ERROR_EXT_POWER_SUPPLY_OFF

LWM2M_DEVICE_ERROR_GPS_FAILURE

LWM2M_DEVICE_ERROR_LOW_SIGNAL_STRENGTH

LWM2M_DEVICE_ERROR_OUT_OF_MEMORY

LWM2M_DEVICE_ERROR_SMS_FAILURE

LWM2M_DEVICE_ERROR_NETWORK_FAILURE

LWM2M_DEVICE_ERROR_PERIPHERAL_FAILURE

LWM2M_DEVICE_BATTERY_STATUS_NORMAL

Battery status codes used for the “Battery Status” resource (3/0/20) of the LwM2M Device object. As the battery status changes, an LwM2M client can set one of the following codes via: lwm2m_engine_set_u8(“3/0/20”, [battery status])

LWM2M_DEVICE_BATTERY_STATUS_CHARGING

LWM2M_DEVICE_BATTERY_STATUS_CHARGE_COMP

LWM2M_DEVICE_BATTERY_STATUS_DAMAGED

LWM2M_DEVICE_BATTERY_STATUS_LOW

LWM2M_DEVICE_BATTERY_STATUS_NOT_INST

LWM2M_DEVICE_BATTERY_STATUS_UNKNOWN

STATE_IDLE

LWM2M Firmware Update object states.

An LwM2M client or the LwM2M Firmware Update object use the following codes to represent the LwM2M Firmware Update state (5/0/3).

STATE_DOWNLOADING

STATE_DOWNLOADED

STATE_UPDATING

RESULT_DEFAULT

LWM2M Firmware Update object result codes.

After processing a firmware update, the client sets the result via one of the following codes via lwm2m_engine_set_u8("5/0/5", [result code])

RESULT_SUCCESS

RESULT_NO_STORAGE

RESULT_OUT_OF_MEM

RESULT_CONNECTION_LOST

RESULT_INTEGRITY_FAILED

RESULT_UNSUP_FW

RESULT_INVALID_URI

RESULT_UPDATE_FAILED

RESULT_UNSUP_PROTO

LWM2M_FLOAT32_DEC_MAX

Data structure used to represent the LwM2M float type: val1 is the whole number portion of the decimal val2 is the decimal portion *1000000 for 32bit, *1000000000 for 64bit Example: 123.456 == val1: 123, val2:456000 Example: 123.000456 = val1: 123, val2:456.

Maximum precision value for 32-bit LwM2M float val2

LWM2M_OBJLNK_MAX_ID

Maximum value for ObjLnk resource fields.

LWM2M_RES_DATA_READ_ONLY

Resource read-only value bit.

LWM2M_RES_DATA_FLAG_RO

Resource read-only flag.

LWM2M_HAS_RES_FLAG(res, f)

Read resource flags helper macro.

LWM2M_RD_CLIENT_FLAG_BOOTSTRAP

Run bootstrap procedure in current session.

Typedefs

typedef void (*lwm2m_socket_fault_cb_t)(int error)

typedef void (*lwm2m_notify_timeout_cb_t)(void)

typedef void *(*lwm2m_engine_get_data_cb_t)(uint16_t obj_inst_id, uint16_t res_id, uint16_t res_inst_id, size_t *data_len)

Asynchronous callback to get a resource buffer and length.

Prior to accessing the data buffer of a resource, the engine can use this callback to get the buffer pointer and length instead of using the resource's data buffer.

The client or LwM2M objects can register a function of this type via: *lwm2m_engine_register_read_callback() lwm2m_engine_register_pre_write_callback()*

Param obj_inst_id **[in]** Object instance ID generating the callback.

Param res_id **[in]** Resource ID generating the callback.

Param res_inst_id **[in]** Resource instance ID generating the callback (typically 0 for non-multi instance resources).

Param data_len **[out]** Length of the data buffer.

Return Callback returns a pointer to the data buffer or NULL for failure.

typedef int (*lwm2m_engine_set_data_cb_t)(uint16_t obj_inst_id, uint16_t res_id, uint16_t res_inst_id, uint8_t *data, uint16_t data_len, bool last_block, size_t total_size)

Asynchronous callback when data has been set to a resource buffer.

After changing the data of a resource buffer, the LwM2M engine can make use of this callback to pass the data back to the client or LwM2M objects.

A function of this type can be registered via: *lwm2m_engine_register_validate_callback() lwm2m_engine_register_post_write_callback()*

Param obj_inst_id **[in]** Object instance ID generating the callback.

Param res_id **[in]** Resource ID generating the callback.

Param res_inst_id **[in]** Resource instance ID generating the callback (typically 0 for non-multi instance resources).

Param data **[in]** Pointer to data.

Param data_len **[in]** Length of the data.

Param last_block **[in]** Flag used during block transfer to indicate the last block of data. For non-block transfers this is always false.

Param total_size **[in]** Expected total size of data for a block transfer. For non-block transfers this is 0.

Return Callback returns a negative error code (errno.h) indicating reason of failure or 0 for success.

typedef int (*lwm2m_engine_user_cb_t)(uint16_t obj_inst_id)

Asynchronous event notification callback.

Various object instance and resource-based events in the LwM2M engine can trigger a callback of this function type: object instance create, and object instance delete.

Register a function of this type via: *lwm2m_engine_register_create_callback() lwm2m_engine_register_delete_callback()*

Param obj_inst_id **[in]** Object instance ID generating the callback.

Return Callback returns a negative error code (errno.h) indicating reason of failure or 0 for success.

typedef int (*`lwm2m_engine_execute_cb_t`)(uint16_t obj_inst_id, uint8_t *args, uint16_t args_len)

Asynchronous execute notification callback.

Resource executes trigger a callback of this type.

Register a function of this type via: *lwm2m_engine_register_exec_callback()*

Param obj_inst_id **[in]** Object instance ID generating the callback.

Param args **[in]** Pointer to execute arguments payload. (This can be NULL if no arguments are provided)

Param args_len **[in]** Length of argument payload in bytes.

Return Callback returns a negative error code (errno.h) indicating reason of failure or 0 for success.

typedef struct *float32_value* `float32_value_t`

32-bit variant of the LwM2M float structure

typedef void (*`lwm2m_ctx_event_cb_t`)(struct *lwm2m_ctx* *ctx, enum *lwm2m_rd_client_event* event)

Asynchronous RD client event callback.

Param ctx **[in]** LwM2M context generating the event

Param event **[in]** LwM2M RD client event code

Enums

enum `lwm2m_rd_client_event`

LwM2M RD client events.

LwM2M client events are passed back to the event_cb function in *lwm2m_rd_client_start()*

Values:

enumerator `LWM2M_RD_CLIENT_EVENT_NONE`

enumerator `LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_FAILURE`

enumerator `LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_COMPLETE`

enumerator `LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_TRANSFER_COMPLETE`

enumerator `LWM2M_RD_CLIENT_EVENT_REGISTRATION_FAILURE`

enumerator `LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE`

enumerator `LWM2M_RD_CLIENT_EVENT_REG_UPDATE_FAILURE`

enumerator LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE

enumerator LWM2M_RD_CLIENT_EVENT_DEREGISTER_FAILURE

enumerator LWM2M_RD_CLIENT_EVENT_DISCONNECT

enumerator LWM2M_RD_CLIENT_EVENT_QUEUE_MODE_RX_OFF

enumerator LWM2M_RD_CLIENT_EVENT_NETWORK_ERROR

Functions

int lwm2m_device_add_err(uint8_t error_code)

Register a new error code with LwM2M Device object.

Parameters

- error_code – **[in]** New error code.

Returns 0 for success or negative in case of error.

int lwm2m_engine_update_observer_min_period(char *pathstr, uint32_t period_s)

Change an observer's pmin value.

LwM2M clients use this function to modify the pmin attribute for an observation being made. Example to update the pmin of a temperature sensor value being observed: lwm2m_engine_update_observer_min_period("3303/0/5700",5);

Parameters

- pathstr – **[in]** LwM2M path string "obj/obj-inst/res"
- period_s – **[in]** Value of pmin to be given (in seconds).

Returns 0 for success or negative in case of error.

int lwm2m_engine_update_observer_max_period(char *pathstr, uint32_t period_s)

Change an observer's pmax value.

LwM2M clients use this function to modify the pmax attribute for an observation being made. Example to update the pmax of a temperature sensor value being observed: lwm2m_engine_update_observer_max_period("3303/0/5700",5);

Parameters

- pathstr – **[in]** LwM2M path string "obj/obj-inst/res"
- period_s – **[in]** Value of pmax to be given (in seconds).

Returns 0 for success or negative in case of error.

int lwm2m_engine_create_obj_inst(char *pathstr)

Create an LwM2M object instance.

LwM2M clients use this function to create non-default LwM2M objects: Example to create first temperature sensor object: lwm2m_engine_create_obj_inst("3303/0");

Parameters

- pathstr – **[in]** LwM2M path string "obj/obj-inst"

Returns 0 for success or negative in case of error.

int `lwm2m_engine_delete_obj_inst`(char *pathstr)

Delete an LwM2M object instance.

LwM2M clients use this function to delete LwM2M objects.

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst"

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_opaque`(char *pathstr, char *data_ptr, uint16_t data_len)

Set resource (instance) value (opaque buffer)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `data_ptr` – **[in]** Data buffer
- `data_len` – **[in]** Length of buffer

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_string`(char *pathstr, char *data_ptr)

Set resource (instance) value (string)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `data_ptr` – **[in]** NULL terminated char buffer

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_u8`(char *pathstr, uint8_t value)

Set resource (instance) value (u8)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** u8 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_u16`(char *pathstr, uint16_t value)

Set resource (instance) value (u16)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** u16 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_u32`(char *pathstr, uint32_t value)

Set resource (instance) value (u32)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** u32 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_u64`(char *pathstr, uint64_t value)

Set resource (instance) value (u64)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** u64 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_s8`(char *pathstr, int8_t value)

Set resource (instance) value (s8)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** s8 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_s16`(char *pathstr, int16_t value)

Set resource (instance) value (s16)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** s16 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_s32`(char *pathstr, int32_t value)

Set resource (instance) value (s32)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** s32 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_s64`(char *pathstr, int64_t value)

Set resource (instance) value (s64)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** s64 value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_bool`(char *pathstr, bool value)

Set resource (instance) value (bool)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** bool value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_float32`(char *pathstr, *float32_value_t* *value)

Set resource (instance) value (32-bit float structure)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"

- `value` – **[in]** 32-bit float value

Returns 0 for success or negative in case of error.

int `lwm2m_engine_set_objlnk`(char *pathstr, struct *lwm2m_objlnk* *value)

Set resource (instance) value (ObjLnk)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[in]** pointer to the *lwm2m_objlnk* structure

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_opaque`(char *pathstr, void *buf, uint16_t buflen)

Get resource (instance) value (opaque buffer)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `buf` – **[out]** Data buffer to copy data into
- `buflen` – **[in]** Length of buffer

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_string`(char *pathstr, void *str, uint16_t strlen)

Get resource (instance) value (string)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `str` – **[out]** String buffer to copy data into
- `strlen` – **[in]** Length of buffer

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_u8`(char *pathstr, uint8_t *value)

Get resource (instance) value (u8)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** u8 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_u16`(char *pathstr, uint16_t *value)

Get resource (instance) value (u16)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** u16 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_u32`(char *pathstr, uint32_t *value)

Get resource (instance) value (u32)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** u32 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_u64`(char *pathstr, uint64_t *value)

Get resource (instance) value (u64)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** u64 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_s8`(char *pathstr, int8_t *value)

Get resource (instance) value (s8)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** s8 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_s16`(char *pathstr, int16_t *value)

Get resource (instance) value (s16)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** s16 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_s32`(char *pathstr, int32_t *value)

Get resource (instance) value (s32)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** s32 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_s64`(char *pathstr, int64_t *value)

Get resource (instance) value (s64)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** s64 buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_bool`(char *pathstr, bool *value)

Get resource (instance) value (bool)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `value` – **[out]** bool buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_float32`(char *pathstr, *float32_value_t* *buf)

Get resource (instance) value (32-bit float structure)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"

- `buf` – **[out]** 32-bit float buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_objlnk`(char *pathstr, struct *lwm2m_objlnk* *buf)

Get resource (instance) value (ObjLnk)

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `buf` – **[out]** *lwm2m_objlnk* buffer to copy data into

Returns 0 for success or negative in case of error.

int `lwm2m_engine_register_read_callback`(char *pathstr, *lwm2m_engine_get_data_cb_t* cb)

Set resource (instance) read callback.

LwM2M clients can use this to set the callback function for resource reads.

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `cb` – **[in]** Read resource callback

Returns 0 for success or negative in case of error.

int `lwm2m_engine_register_pre_write_callback`(char *pathstr, *lwm2m_engine_get_data_cb_t* cb)

Set resource (instance) pre-write callback.

This callback is triggered before setting the value of a resource. It can pass a special data buffer to the engine so that the actual resource value can be calculated later, etc.

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `cb` – **[in]** Pre-write resource callback

Returns 0 for success or negative in case of error.

int `lwm2m_engine_register_validate_callback`(char *pathstr, *lwm2m_engine_set_data_cb_t* cb)

Set resource (instance) validation callback.

This callback is triggered before setting the value of a resource to the resource data buffer.

The callback allows an LwM2M client or object to validate the data before writing and notify an error if the data should be discarded for any reason (by returning a negative error code).

Note: All resources that have a validation callback registered are initially decoded into a temporary validation buffer. Make sure that `CONFIG_LWM2M_ENGINE_VALIDATION_BUFFER_SIZE` is large enough to store each of the validated resources (individually).

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `cb` – **[in]** Validate resource data callback

Returns 0 for success or negative in case of error.

int lwm2m_engine_register_post_write_callback(char *pathstr, *lwm2m_engine_set_data_cb_t* cb)

Set resource (instance) post-write callback.

This callback is triggered after setting the value of a resource to the resource data buffer.

It allows an LwM2M client or object to post-process the value of a resource or trigger other related resource calculations.

Parameters

- pathstr – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- cb – **[in]** Post-write resource callback

Returns 0 for success or negative in case of error.

int lwm2m_engine_register_exec_callback(char *pathstr, *lwm2m_engine_execute_cb_t* cb)

Set resource execute event callback.

This event is triggered when the execute method of a resource is enabled.

Parameters

- pathstr – **[in]** LwM2M path string "obj/obj-inst/res"
- cb – **[in]** Execute resource callback

Returns 0 for success or negative in case of error.

int lwm2m_engine_register_create_callback(uint16_t obj_id, *lwm2m_engine_user_cb_t* cb)

Set object instance create event callback.

This event is triggered when an object instance is created.

Parameters

- obj_id – **[in]** LwM2M object id
- cb – **[in]** Create object instance callback

Returns 0 for success or negative in case of error.

int lwm2m_engine_register_delete_callback(uint16_t obj_id, *lwm2m_engine_user_cb_t* cb)

Set object instance delete event callback.

This event is triggered when an object instance is deleted.

Parameters

- obj_id – **[in]** LwM2M object id
- cb – **[in]** Delete object instance callback

Returns 0 for success or negative in case of error.

int lwm2m_engine_set_res_data(char *pathstr, void *data_ptr, uint16_t data_len, uint8_t data_flags)

Set data buffer for a resource.

Use this function to set the data buffer and flags for the specified LwM2M resource.

Parameters

- pathstr – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- data_ptr – **[in]** Data buffer pointer
- data_len – **[in]** Length of buffer
- data_flags – **[in]** Data buffer flags (such as read-only, etc)

Returns 0 for success or negative in case of error.

int `lwm2m_engine_get_res_data`(char *pathstr, void **data_ptr, uint16_t *data_len, uint8_t *data_flags)

Get data buffer for a resource.

Use this function to get the data buffer information for the specified LwM2M resource.

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res(/res-inst)"
- `data_ptr` – **[out]** Data buffer pointer
- `data_len` – **[out]** Length of buffer
- `data_flags` – **[out]** Data buffer flags (such as read-only, etc)

Returns 0 for success or negative in case of error.

int `lwm2m_engine_create_res_inst`(char *pathstr)

Create a resource instance.

LwM2M clients use this function to create multi-resource instances: Example to create 0 instance of device available power sources: lwm2m_engine_create_res_inst("3/0/6/0");

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res/res-inst"

Returns 0 for success or negative in case of error.

int `lwm2m_engine_delete_res_inst`(char *pathstr)

Delete a resource instance.

Use this function to remove an existing resource instance

Parameters

- `pathstr` – **[in]** LwM2M path string "obj/obj-inst/res/res-inst"

Returns 0 for success or negative in case of error.

int `lwm2m_engine_update_service_period`(*k_work_handler_t* service, uint32_t period_ms)

Update the period of a given service.

Allow the period modification on an existing service created with lwm2m_engine_add_service(). Example to frequency at which a periodic_service changes it's values : lwm2m_engine_update_service(device_periodic_service,5*MSEC_PER_SEC);

Parameters

- `service` – **[in]** Handler of the periodic_service
- `period_ms` – **[in]** New period for the periodic_service (in milliseconds)

Returns 0 for success or negative in case of error.

int `lwm2m_engine_start`(struct *lwm2m_ctx* *client_ctx)

Start the LwM2M engine.

LwM2M clients normally do not need to call this function as it is called by *lwm2m_rd_client_start()*. However, if the client does not use the RD client implementation, it will need to be called manually.

Parameters

- `client_ctx` – **[in]** LwM2M context

Returns 0 for success or negative in case of error.

void `lwm2m_acknowledge`(struct *lwm2m_ctx* *client_ctx)

Acknowledge the currently processed request with an empty ACK.

LwM2M engine by default sends piggybacked responses for requests. This function allows to send an empty ACK for a request earlier (from the application callback). The LwM2M engine will then send the actual response as a separate CON message after all callbacks are executed.

Parameters

- `client_ctx` – **[in]** LwM2M context

void `lwm2m_rd_client_start`(struct *lwm2m_ctx* *client_ctx, const char *ep_name, uint32_t flags, *lwm2m_ctx_event_cb_t* event_cb)

Start the LwM2M RD (Registration / Discovery) Client.

The RD client sits just above the LwM2M engine and performs the necessary actions to implement the "Registration interface". For more information see Section 5.3 "Client Registration Interface" of the LwM2M Technical Specification.

NOTE: *lwm2m_engine_start()* is called automatically by this function.

Parameters

- `client_ctx` – **[in]** LwM2M context
- `ep_name` – **[in]** Registered endpoint name
- `flags` – **[in]** Flags used to configure current LwM2M session.
- `event_cb` – **[in]** Client event callback function

void `lwm2m_rd_client_stop`(struct *lwm2m_ctx* *client_ctx, *lwm2m_ctx_event_cb_t* event_cb)

Stop the LwM2M RD (De-register) Client.

The RD client sits just above the LwM2M engine and performs the necessary actions to implement the "Registration interface". For more information see Section 5.3 "Client Registration Interface" of the LwM2M Technical Specification.

Parameters

- `client_ctx` – **[in]** LwM2M context
- `event_cb` – **[in]** Client event callback function

void `lwm2m_rd_client_update`(void)

Trigger a Registration Update of the LwM2M RD Client.

struct `lwm2m_ctx`

#include <lwm2m.h> LwM2M context structure to maintain information for a single LwM2M connection.

Public Members

struct *sockaddr* `remote_addr`

Destination address storage

struct *coap_pending* `pendings`[CONFIG_LWM2M_ENGINE_MAX_PENDING]

Private CoAP and networking structures

void *processed_req

A pointer to currently processed request, for internal LwM2M engine use. The underlying type is struct lwm2m_message, but since it's declared in a private header and not exposed to the application, it's stored as a void pointer.

bool use_dtls

Flag to indicate if context should use DTLS. Enabled via the use of coaps:// protocol prefix in connection information. NOTE: requires CONFIG_LWM2M_DTLS_SUPPORT=y

int sec_obj_inst

Current index of Security Object used for server credentials

int srv_obj_inst

Current index of Server Object used in this context.

bool bootstrap_mode

Flag to enable BOOTSTRAP interface. See Section 5.2 "Bootstrap Interface" of LwM2M Technical Specification 1.0.2 for more information.

int sock_fd

Socket File Descriptor

lwm2m_socket_fault_cb_t fault_cb

Socket fault callback. LwM2M processing thread will call this callback in case of socket errors on receive.

lwm2m_notify_timeout_cb_t notify_timeout_cb

Notify Timeout Callback. LwM2M processing thread will call this callback in case of notify timeout.

uint8_t validate_buf[CONFIG_LWM2M_ENGINE_VALIDATION_BUFFER_SIZE]

Validation buffer. Used as a temporary buffer to decode the resource value before validation. On successful validation, its content is copied into the actual resource buffer.

struct float32_value

#include <lwm2m.h> 32-bit variant of the LwM2M float structure

struct lwm2m_objlnk

#include <lwm2m.h> LWM2M ObjLnk resource type structure.

MQTT

Overview MQTT (Message Queuing Telemetry Transport) is an application layer protocol which works on top of the TCP/IP stack. It is a lightweight publish/subscribe messaging transport for machine-to-machine communication. For more information about the protocol itself, see http://mqtt.org/.

Zephyr provides an MQTT client library built on top of BSD sockets API. The library is configurable at a per-client basis, with support for MQTT versions 3.1.0 and 3.1.1. The Zephyr MQTT implementation can be used with either plain sockets communicating over TCP, or with secure sockets communicating over TLS. See *BSD Sockets* for more information about Zephyr sockets.

MQTT clients require an MQTT server to connect to. Such a server, called an MQTT Broker, is responsible for managing client subscriptions and distributing messages published by clients. There are many implementations of MQTT brokers, one of them being Eclipse Mosquitto. See https://mosquitto.org/ for more information about the Eclipse Mosquitto project.

Sample usage To create an MQTT client, a client context structure and buffers need to be defined:

```
/* Buffers for MQTT client. */
static uint8_t rx_buffer[256];
static uint8_t tx_buffer[256];

/* MQTT client context */
static struct mqtt_client client_ctx;
```

Multiple MQTT client instances can be created in the application and managed independently. Additionally, a structure for MQTT Broker address information is needed. This structure must be accessible throughout the lifespan of the MQTT client and can be shared among MQTT clients:

```
/* MQTT Broker address information. */
static struct sockaddr_storage broker;
```

An MQTT client library will notify MQTT events to the application through a callback function created to handle respective events:

```
void mqtt_evt_handler(struct mqtt_client *client,
                      const struct mqtt_evt *evt)
{
   switch (evt->type) {
      /* Handle events here. */
   }
}
```

For a list of possible events, see *API Reference*.

The client context structure needs to be initialized and set up before it can be used. An example configuration for TCP transport is shown below:

```
mqtt_client_init(&client_ctx);

/* MQTT client configuration */
client_ctx.broker = &broker;
client_ctx.evt_cb = mqtt_evt_handler;
client_ctx.client_id.utf8 = (uint8_t *)"zephyr_mqtt_client";
client_ctx.client_id.size = sizeof("zephyr_mqtt_client") - 1;
client_ctx.password = NULL;
client_ctx.user_name = NULL;
client_ctx.protocol_version = MQTT_VERSION_3_1_1;
client_ctx.transport.type = MQTT_TRANSPORT_NON_SECURE;

/* MQTT buffers configuration */
```

(continues on next page)

(continued from previous page)

```
client_ctx.rx_buf = rx_buffer;
client_ctx.rx_buf_size = sizeof(rx_buffer);
client_ctx.tx_buf = tx_buffer;
client_ctx.tx_buf_size = sizeof(tx_buffer);
```

After the configuration is set up, the MQTT client can connect to the MQTT broker. Call the `mqtt_connect` function, which will create the appropriate socket, establish a TCP/TLS connection, and send an `MQTT CONNECT` message. When notified, the application should call the `mqtt_input` function to process the response received. Note, that `mqtt_input` is a non-blocking function, therefore the application should use socket `poll` to wait for the response. If the connection was successful, `MQTT_EVT_CONNACK` will be notified to the application through the callback function.

```
rc = mqtt_connect(&client_ctx);
if (rc != 0) {
   return rc;
}

fds[0].fd = client_ctx.transport.tcp.sock;
fds[0].events = ZSOCK_POLLIN;
poll(fds, 1, K_MSEC(5000));

mqtt_input(&client_ctx);

if (!connected) {
   mqtt_abort(&client_ctx);
}
```

In the above code snippet, the MQTT callback function should set the `connected` flag upon a successful connection. If the connection fails at the MQTT level or a timeout occurs, the connection will be aborted, and the underlying socket closed.

After the connection is established, an application needs to call `mqtt_input` and `mqtt_live` functions periodically to process incoming data and upkeep the connection. If an MQTT message is received, an MQTT callback function will be called and an appropriate event notified.

The connection can be closed by calling the `mqtt_disconnect` function.

Zephyr provides sample code utilizing the MQTT client API. See mqtt-publisher-sample for more information.

Using MQTT with TLS The Zephyr MQTT library can be used with TLS transport for secure communication by selecting a secure transport type (`MQTT_TRANSPORT_SECURE`) and some additional configuration information:

```
client_ctx.transport.type = MQTT_TRANSPORT_SECURE;

struct mqtt_sec_config *tls_config = &client_ctx.transport.tls.config;

tls_config->peer_verify = TLS_PEER_VERIFY_REQUIRED;
tls_config->cipher_list = NULL;
tls_config->sec_tag_list = m_sec_tags;
tls_config->sec_tag_count = ARRAY_SIZE(m_sec_tags);
tls_config->hostname = MQTT_BROKER_HOSTNAME;
```

In this sample code, the `m_sec_tags` array holds a list of tags, referencing TLS credentials that the MQTT library should use for authentication. We do not specify `cipher_list`, to allow the use of all cipher suites available in the system. We set `hostname` field to broker hostname, which is required for server authentication. Finally, we enforce peer certificate verification by setting the `peer_verify` field.

Note, that TLS credentials referenced by the `m_sec_tags` array must be registered in the system first. For more information on how to do that, refer to *secure sockets documentation*.

An example of how to use TLS with MQTT is also present in mqtt-publisher-sample.

API Reference

group `mqtt_socket`

MQTT Client Implementation.

MQTT Client's Application interface is defined in this header.

Note: The implementation assumes TCP module is enabled.

Note: By default the implementation uses MQTT version 3.1.1.

Defines

`MQTT_UTF8_LITERAL`(literal)

Initialize UTF-8 encoded string from C literal string.

Use it as follows:

struct *mqtt_utf8* password = MQTT_UTF8_LITERAL("my_pass");

Parameters

- `literal` – **[in]** Literal string from which to generate *mqtt_utf8* object.

Typedefs

typedef void (*`mqtt_evt_cb_t`)(struct *mqtt_client* *client, const struct *mqtt_evt* *evt)

Asynchronous event notification callback registered by the application.

Param client **[in]** Identifies the client for which the event is notified.

Param evt **[in]** Event description along with result and associated parameters (if any).

Enums

enum `mqtt_evt_type`

MQTT Asynchronous Events notified to the application from the module through the callback registered by the application.

Values:

enumerator `MQTT_EVT_CONNACK`

Acknowledgment of connection request. Event result accompanying the event indicates whether the connection failed or succeeded.

enumerator MQTT_EVT_DISCONNECT

Disconnection Event. MQTT Client Reference is no longer valid once this event is received for the client.

enumerator MQTT_EVT_PUBLISH

Publish event received when message is published on a topic client is subscribed to.

Note: PUBLISH event structure only contains payload size, the payload data parameter should be ignored. Payload content has to be read manually with *mqtt_read_publish_payload* function.

---

enumerator MQTT_EVT_PUBACK

Acknowledgment for published message with QoS 1.

enumerator MQTT_EVT_PUBREC

Reception confirmation for published message with QoS 2.

enumerator MQTT_EVT_PUBREL

Release of published message with QoS 2.

enumerator MQTT_EVT_PUBCOMP

Confirmation to a publish release message with QoS 2.

enumerator MQTT_EVT_SUBACK

Acknowledgment to a subscribe request.

enumerator MQTT_EVT_UNSUBACK

Acknowledgment to a unsubscribe request.

enumerator MQTT_EVT_PINGRESP

Ping Response from server.

enum mqtt_version

MQTT version protocol level.

Values:

enumerator MQTT_VERSION_3_1_0 = 3

Protocol level for 3.1.0.

enumerator MQTT_VERSION_3_1_1 = 4

Protocol level for 3.1.1.

enum mqtt_qos

MQTT Quality of Service types.

Values:

enumerator MQTT_QOS_0_AT_MOST_ONCE = 0x00

Lowest Quality of Service, no acknowledgment needed for published message.

enumerator MQTT_QOS_1_AT_LEAST_ONCE = 0x01

Medium Quality of Service, if acknowledgment expected for published message, duplicate messages permitted.

enumerator MQTT_QOS_2_EXACTLY_ONCE = 0x02

Highest Quality of Service, acknowledgment expected and message shall be published only once. Message not published to interested parties unless client issues a PUBREL.

enum mqtt_conn_return_code

MQTT CONNACK return codes.

Values:

enumerator MQTT_CONNECTION_ACCEPTED = 0x00

Connection accepted.

enumerator MQTT_UNACCEPTABLE_PROTOCOL_VERSION = 0x01

The Server does not support the level of the MQTT protocol requested by the Client.

enumerator MQTT_IDENTIFIER_REJECTED = 0x02

The Client identifier is correct UTF-8 but not allowed by the Server.

enumerator MQTT_SERVER_UNAVAILABLE = 0x03

The Network Connection has been made but the MQTT service is unavailable.

enumerator MQTT_BAD_USER_NAME_OR_PASSWORD = 0x04

The data in the user name or password is malformed.

enumerator MQTT_NOT_AUTHORIZED = 0x05

The Client is not authorized to connect.

enum mqtt_suback_return_code

MQTT SUBACK return codes.

Values:

enumerator MQTT_SUBACK_SUCCESS_QoS_0 = 0x00

Subscription with QoS 0 succeeded.

enumerator MQTT_SUBACK_SUCCESS_QoS_1 = 0x01

Subscription with QoS 1 succeeded.

enumerator MQTT_SUBACK_SUCCESS_QoS_2 = 0x02

Subscription with QoS 2 succeeded.

enumerator MQTT_SUBACK_FAILURE = 0x80

Subscription for a topic failed.

enum mqtt_transport_type

MQTT transport type.

Values:

enumerator MQTT_TRANSPORT_NON_SECURE

Use non secure TCP transport for MQTT connection.

enumerator MQTT_TRANSPORT_NUM

Shall not be used as a transport type. Indicator of maximum transport types possible.

Functions

void mqtt_client_init(struct *mqtt_client* *client)

Initializes the client instance.

Note: Shall be called to initialize client structure, before setting any client parameters and before connecting to broker.

Parameters

- client – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

int mqtt_connect(struct *mqtt_client* *client)

API to request new MQTT client connection.

Note: This memory is assumed to be resident until mqtt_disconnect is called.

Note: Any subsequent changes to parameters like broker address, user name, device id, etc. have no effect once MQTT connection is established.

Note: Default protocol revision used for connection request is 3.1.1. Please set client.protocol_version = MQTT_VERSION_3_1_0 to use protocol 3.1.0.

Note: Please modify CONFIG_MQTT_KEEPALIVE time to override default of 1 minute.

Parameters

- client – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int mqtt_publish(struct *mqtt_client* *client, const struct *mqtt_publish_param* *param)

API to publish messages on topics.

Parameters

- client – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- param – **[in]** Parameters to be used for the publish message. Shall not be NULL.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_publish_qos1_ack`(struct *mqtt_client* *client, const struct *mqtt_puback_param* *param)

API used by client to send acknowledgment on receiving QoS1 publish message. Should be called on reception of *MQTT_EVT_PUBLISH* with QoS level *MQTT_QOS_1_AT_LEAST_ONCE*.

Parameters

- `client` – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Identifies message being acknowledged.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_publish_qos2_receive`(struct *mqtt_client* *client, const struct *mqtt_pubrec_param* *param)

API used by client to send acknowledgment on receiving QoS2 publish message. Should be called on reception of *MQTT_EVT_PUBLISH* with QoS level *MQTT_QOS_2_EXACTLY_ONCE*.

Parameters

- `client` – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Identifies message being acknowledged.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_publish_qos2_release`(struct *mqtt_client* *client, const struct *mqtt_pubrel_param* *param)

API used by client to request release of QoS2 publish message. Should be called on reception of *MQTT_EVT_PUBREC*.

Parameters

- `client` – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Identifies message being released.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_publish_qos2_complete`(struct *mqtt_client* *client, const struct *mqtt_pubcomp_param* *param)

API used by client to send acknowledgment on receiving QoS2 publish release message. Should be called on reception of *MQTT_EVT_PUBREL*.

Parameters

- `client` – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Identifies message being completed.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_subscribe`(struct *mqtt_client* *client, const struct *mqtt_subscription_list* *param)

API to request subscription of one or more topics on the connection.

Parameters

- `client` – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Subscription parameters. Shall not be NULL.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_unsubscribe`(struct *mqtt_client* *client, const struct *mqtt_subscription_list* *param)

API to request unsubscription of one or more topics on the connection.

Note: QoS included in topic description is unused in this API.

Parameters

- `client` – **[in]** Identifies client instance for which the procedure is requested. Shall not be NULL.
- `param` – **[in]** Parameters describing topics being unsubscribed from. Shall not be NULL.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_ping`(struct *mqtt_client* *client)

API to send MQTT ping. The use of this API is optional, as the library handles the connection keep-alive on it's own, see *mqtt_live*.

Parameters

- `client` – **[in]** Identifies client instance for which procedure is requested.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_disconnect`(struct *mqtt_client* *client)

API to disconnect MQTT connection.

Parameters

- `client` – **[in]** Identifies client instance for which procedure is requested.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_abort`(struct *mqtt_client* *client)

API to abort MQTT connection. This will close the corresponding transport without closing the connection gracefully at the MQTT level (with disconnect message).

Parameters

- `client` – **[in]** Identifies client instance for which procedure is requested.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int `mqtt_live`(struct *mqtt_client* *client)

This API should be called periodically for the client to be able to keep the connection alive by sending Ping Requests if need be.

Note: Application shall ensure that the periodicity of calling this function makes it possible to respect the Keep Alive time agreed with the broker on connection. *mqtt_connect* for details on Keep Alive time.

Parameters

- `client` – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int mqtt_keepalive_time_left(const struct *mqtt_client* *client)

Helper function to determine when next keep alive message should be sent. Can be used for instance as a source for poll timeout.

Parameters

- client – **[in]** Client instance for which the procedure is requested.

Returns Time in milliseconds until next keep alive message is expected to be sent. Function will return -1 if keep alive messages are not enabled.

int mqtt_input(struct *mqtt_client* *client)

Receive an incoming MQTT packet. The registered callback will be called with the packet content.

Note: In case of PUBLISH message, the payload has to be read separately with *mqtt_read_publish_payload* function. The size of the payload to read is provided in the publish event structure.

Note: This is a non-blocking call.

Parameters

- client – **[in]** Client instance for which the procedure is requested. Shall not be NULL.

Returns 0 or a negative error code (errno.h) indicating reason of failure.

int mqtt_read_publish_payload(struct *mqtt_client* *client, void *buffer, size_t length)

Read the payload of the received PUBLISH message. This function should be called within the MQTT event handler, when MQTT PUBLISH message is notified.

Note: This is a non-blocking call.

Parameters

- client – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- buffer – **[out]** Buffer where payload should be stored.
- length – **[in]** Length of the buffer, in bytes.

Returns Number of bytes read or a negative error code (errno.h) indicating reason of failure.

int mqtt_read_publish_payload_blocking(struct *mqtt_client* *client, void *buffer, size_t length)

Blocking version of *mqtt_read_publish_payload* function.

Parameters

- client – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- buffer – **[out]** Buffer where payload should be stored.
- length – **[in]** Length of the buffer, in bytes.

Returns Number of bytes read or a negative error code (errno.h) indicating reason of failure.

int `mqtt_readall_publish_payload`(struct *mqtt_client* *client, uint8_t *buffer, size_t length)

Blocking version of *mqtt_read_publish_payload* function which runs until the required number of bytes are read.

Parameters

- `client` – **[in]** Client instance for which the procedure is requested. Shall not be NULL.
- `buffer` – **[out]** Buffer where payload should be stored.
- `length` – **[in]** Number of bytes to read.

Returns 0 if success, otherwise a negative error code (errno.h) indicating reason of failure.

struct `mqtt_utf8`

#include <mqtt.h> Abstracts UTF-8 encoded strings.

Public Members

const uint8_t *`utf8`

Pointer to UTF-8 string.

uint32_t `size`

Size of UTF string, in bytes.

struct `mqtt_binstr`

#include <mqtt.h> Abstracts binary strings.

Public Members

uint8_t *`data`

Pointer to binary stream.

uint32_t `len`

Length of binary stream.

struct `mqtt_topic`

#include <mqtt.h> Abstracts MQTT UTF-8 encoded topic that can be subscribed to or published.

Public Members

struct *mqtt_utf8* `topic`

Topic on to be published or subscribed to.

uint8_t `qos`

Quality of service requested for the subscription. *mqtt_qos* for details.

struct `mqtt_publish_message`

#include <mqtt.h> Parameters for a publish message.

Public Members

struct *mqtt_topic* `topic`

Topic on which data was published.

struct *mqtt_binstr* `payload`

Payload on the topic published.

struct `mqtt_connack_param`

#include <*mqtt.h*> Parameters for a connection acknowledgment (CONNACK).

Public Members

uint8_t `session_present_flag`

The Session Present flag enables a Client to establish whether the Client and Server have a consistent view about whether there is already stored Session state.

enum *mqtt_conn_return_code* `return_code`

The appropriate non-zero Connect return code indicates if the Server is unable to process a connection request for some reason.

struct `mqtt_puback_param`

#include <*mqtt.h*> Parameters for MQTT publish acknowledgment (PUBACK).

struct `mqtt_pubrec_param`

#include <*mqtt.h*> Parameters for MQTT publish receive (PUBREC).

struct `mqtt_pubrel_param`

#include <*mqtt.h*> Parameters for MQTT publish release (PUBREL).

struct `mqtt_pubcomp_param`

#include <*mqtt.h*> Parameters for MQTT publish complete (PUBCOMP).

struct `mqtt_suback_param`

#include <*mqtt.h*> Parameters for MQTT subscription acknowledgment (SUBACK).

struct `mqtt_unsuback_param`

#include <*mqtt.h*> Parameters for MQTT unsubscribe acknowledgment (UNSUBACK).

struct `mqtt_publish_param`

#include <*mqtt.h*> Parameters for a publish message.

Public Members

struct *mqtt_publish_message* `message`

Messages including topic, QoS and its payload (if any) to be published.

uint16_t `message_id`

Message id used for the publish message. Redundant for QoS 0.

uint8_t `dup_flag`

Duplicate flag. If 1, it indicates the message is being retransmitted. Has no meaning with QoS 0.

uint8_t `retain_flag`

Retain flag. If 1, the message shall be stored persistently by the broker.

struct `mqtt_subscription_list`

#include <mqtt.h> List of topics in a subscription request.

Public Members

struct *mqtt_topic* *`list`

Array containing topics along with QoS for each.

uint16_t `list_count`

Number of topics in the subscription list

uint16_t `message_id`

Message id used to identify subscription request.

union `mqtt_evt_param`

#include <mqtt.h> Defines event parameters notified along with asynchronous events to the application.

Public Members

struct *mqtt_connack_param* `connack`

Parameters accompanying MQTT_EVT_CONNACK event.

struct *mqtt_publish_param* `publish`

Parameters accompanying MQTT_EVT_PUBLISH event.

Note: PUBLISH event structure only contains payload size, the payload data parameter should be ignored. Payload content has to be read manually with *mqtt_read_publish_payload* function.

---

struct *mqtt_puback_param* `puback`

Parameters accompanying MQTT_EVT_PUBACK event.

struct *mqtt_pubrec_param* `pubrec`

Parameters accompanying MQTT_EVT_PUBREC event.

struct *mqtt_pubrel_param* `pubrel`

Parameters accompanying MQTT_EVT_PUBREL event.

struct *mqtt_pubcomp_param* `pubcomp`

Parameters accompanying MQTT_EVT_PUBCOMP event.

struct *mqtt_suback_param* `suback`

Parameters accompanying MQTT_EVT_SUBACK event.

struct *mqtt_unsuback_param* `unsuback`

Parameters accompanying MQTT_EVT_UNSUBACK event.

struct `mqtt_evt`

#include <mqtt.h> Defines MQTT asynchronous event notified to the application.

Public Members

enum *mqtt_evt_type* `type`

Identifies the event.

union *mqtt_evt_param* `param`

Contains parameters (if any) accompanying the event.

int `result`

Event result. 0 or a negative error code (errno.h) indicating reason of failure.

struct `mqtt_sec_config`

#include <mqtt.h> TLS configuration for secure MQTT transports.

Public Members

int `peer_verify`

Indicates the preference for peer verification.

uint32_t `cipher_count`

Indicates the number of entries in the cipher list.

int *`cipher_list`

Indicates the list of ciphers to be used for the session. May be NULL to use the default ciphers.

uint32_t `sec_tag_count`

Indicates the number of entries in the sec tag list.

sec_tag_t *`sec_tag_list`

Indicates the list of security tags to be used for the session.

const char *hostname

Peer hostname for ceritificate verification. May be NULL to skip hostname verification.

struct mqtt_transport

#include <mqtt.h> MQTT transport specific data.

Public Members

enum *mqtt_transport_type* type

Transport type selection for client instance. *mqtt_transport_type* for possible values. MQTT_TRANSPORT_MAX is not a valid type.

int sock

Socket descriptor.

struct mqtt_internal

#include <mqtt.h> MQTT internal state.

Public Members

struct sys_mutex mutex

Internal. Mutex to protect access to the client instance.

uint32_t last_activity

Internal. Wall clock value (in milliseconds) of the last activity that occurred. Needed for periodic PING.

uint32_t state

Internal. Client's state in the connection.

uint32_t rx_buf_datalen

Internal. Packet length read so far.

uint32_t remaining_payload

Internal. Remaining payload length to read.

struct mqtt_client

#include <mqtt.h> MQTT Client definition to maintain information relevant to the client.

Public Members

struct *mqtt_internal* internal

MQTT client internal state.

struct *mqtt_transport* transport

MQTT transport configuration and data.

struct *mqtt_utf8* `client_id`

Unique client identification to be used for the connection.

const void *`broker`

Broker details, for example, address, port. Address type should be compatible with transport used.

struct *mqtt_utf8* *`user_name`

User name (if any) to be used for the connection. NULL indicates no user name.

struct *mqtt_utf8* *`password`

Password (if any) to be used for the connection. Note that if password is provided, user name shall also be provided. NULL indicates no password.

struct *mqtt_topic* *`will_topic`

Will topic and QoS. Can be NULL.

struct *mqtt_utf8* *`will_message`

Will message. Can be NULL. Non NULL value valid only if will topic is not NULL.

mqtt_evt_cb_t `evt_cb`

Application callback registered with the module to get MQTT events.

uint8_t *`rx_buf`

Receive buffer used for MQTT packet reception in RX path.

uint32_t `rx_buf_size`

Size of receive buffer.

uint8_t *`tx_buf`

Transmit buffer used for creating MQTT packet in TX path.

uint32_t `tx_buf_size`

Size of transmit buffer.

uint16_t `keepalive`

Keepalive interval for this client in seconds. Default is CONFIG_MQTT_KEEPALIVE.

uint8_t `protocol_version`

MQTT protocol version.

int8_t `unacked_ping`

Unanswered PINGREQ count on this connection.

uint8_t `will_retain`

Will retain flag, 1 if will message shall be retained persistently.

uint8_t `clean_session`

Clean session flag indicating a fresh (1) or a retained session (0). Default is CONFIG_MQTT_CLEAN_SESSION.

7.20.5 Network System Management

Network Configuration Library

Overview The network configuration library sets up networking devices in a semi-automatic way during the system boot, based on user-supplied Kconfig options.

The following Kconfig options affect how configuration library will setup the system:

Table 5: Kconfig options for network configuration library

Option name	Description
CONFIG_NET_CONFIG_SETTINGS	This option controls whether the network system is configured or initialized at all. If not set, then the config library is not used for initialization and the application needs to do all the network related configuration itself. If this option is set, then the user can optionally configure static IP addresses to be set to the first network interface in the system. Typically setting static IP addresses is only usable in testing and should not be used in production code. See the config library Kconfig file subsys/net/lib/config/Kconfig for specific options to set the static IP addresses.
CONFIG_NET_CONFIG_AUTO_INIT	The networking system is automatically configured when the device is started.
CONFIG_NET_CONFIG_INIT_TIMEOUT	This configures how long to wait for the networking to be ready and available. If for example IPv4 address from DHCPv4 is not received within this limit, then a call to `net_config_init()` will return error during the device startup.
CONFIG_NET_CONFIG_NEED_IPV4	The network application needs IPv4 support to function properly. This option makes sure the network application is initialized properly in order to use IPv4. If `CONFIG_NET_IPV4` is not enabled, then setting this option will automatically enable IPv4.
CONFIG_NET_CONFIG_NEED_IPV6	The network application needs IPv6 support to function properly. This option makes sure the network application is initialized properly in order to use IPv6. If `CONFIG_NET_IPV6` is not enabled, then setting this option will automatically enable IPv6.
CONFIG_NET_CONFIG_NEED_ROUTER	If IPv6 is enabled, then this option tells that the network application needs IPv6 router to exists before continuing. This means in practice that the application wants to wait until it receives IPv6 router advertisement message before continuing.
CONFIG_NET_CONFIG_BT_NODE	Enables application to operate in Bluetooth node mode which requires GATT service to be registered and start advertising as peripheral.

Sample usage If `CONFIG_NET_CONFIG_AUTO_INIT` is set, then the configuration library is automatically enabled and run during the device boot. In this case, the library will call `net_config_init()` automatically and the application does not need to do any network configuration.

If you want to use the network configuration library but without automatic initialization, you can call `net_config_init()` manually. The `flags` parameter can be used to give hints to the library about what kind of functionality the application wishes to have before the actual application starts.

API Reference

group `net_config`

Network configuration library.

Defines

NET_CONFIG_NEED_ROUTER

Application needs routers to be set so that connectivity to remote network is possible. For IPv6 networks, this means that the device should receive IPv6 router advertisement message before continuing.

NET_CONFIG_NEED_IPV6

Application needs IPv6 subsystem configured and initialized. Typically this means that the device has IPv6 address set.

NET_CONFIG_NEED_IPV4

Application needs IPv4 subsystem configured and initialized. Typically this means that the device has IPv4 address set.

Functions

int net_config_init(const char *app_info, uint32_t flags, int32_t timeout)

Initialize this network application.

This will call *net_config_init_by_iface()* with NULL network interface.

Parameters

- app_info – String describing this application.
- flags – Flags related to services needed by the client.
- timeout – How long to wait the network setup before continuing the startup.

Returns 0 if ok, <0 if error.

int net_config_init_by_iface(struct *net_if* *iface, const char *app_info, uint32_t flags, int32_t timeout)

Initialize this network application using a specific network interface.

If network interface is set to NULL, then the default one is used in the configuration.

Parameters

- iface – Initialize networking using this network interface.
- app_info – String describing this application.
- flags – Flags related to services needed by the client.
- timeout – How long to wait the network setup before continuing the startup.

Returns 0 if ok, <0 if error.

int net_config_init_app(const struct *device* *dev, const char *app_info)

Initialize this network application.

If CONFIG_NET_CONFIG_AUTO_INIT is set, then this function is called automatically when the device boots. If that is not desired, unset the config option and call the function manually when the application starts.

Parameters

- dev – Network device to use. The function will figure out what network interface to use based on the device. If the device is NULL, then default network interface is used by the function.
- app_info – String describing this application.

Returns 0 if ok, <0 if error.

DHCPv4

Overview The Dynamic Host Configuration Protocol (DHCP) is a network management protocol used on IPv4 networks. A DHCPv4 server dynamically assigns an IPv4 address and other network configuration parameters to each device on a network so they can communicate with other IP networks. See this DHCP Wikipedia article for a detailed overview of how DHCP works.

Note that Zephyr only supports DHCP client functionality.

Sample usage See dhcpv4-client-sample for details.

API Reference

group `dhcpv4`

DHCPv4.

Functions

void `net_dhcpv4_start`(struct *net_if* *iface)

Start DHCPv4 client on an iface.

Start DHCPv4 client on a given interface. DHCPv4 client will start negotiation for IPv4 address. Once the negotiation is success IPv4 address details will be added to interface.

Parameters

- `iface` – A valid pointer on an interface

void `net_dhcpv4_stop`(struct *net_if* *iface)

Stop DHCPv4 client on an iface.

Stop DHCPv4 client on a given interface. DHCPv4 client will remove all configuration obtained from a DHCP server from the interface and stop any further negotiation with the server.

Parameters

- `iface` – A valid pointer on an interface

Hostname Configuration

Overview A networked device might need a hostname, for example, if the device is configured to be a mDNS responder (see *DNS Resolve* for details) and needs to respond to `<hostname>.local` DNS queries.

The `CONFIG_NET_HOSTNAME_ENABLE` must be set in order to store the hostname and enable the relevant APIs. If the option is enabled, then the default hostname is set to be `zephyr` by `CONFIG_NET_HOSTNAME` option.

If the same firmware image is used to flash multiple boards, then it is not practical to use the same hostname in all of the boards. In that case, one can enable `CONFIG_NET_HOSTNAME_UNIQUE` which will add a unique postfix to the hostname. By default the link local address of the first network interface is used as a postfix. In Ethernet networks, the link local address refers to MAC address. For example, if the link local address is `01:02:03:04:05:06`, then the unique hostname could be `zephyr010203040506`. If you want to set the prefix yourself, then call `net_hostname_set_postfix()` before the network interfaces are created. For example for the Ethernet networks, the initialization priority is set by `CONFIG_ETH_INIT_PRIORITY` so you would need to set the postfix before that. The postfix can be set only once.

API Reference

group `net_hostname`

Network hostname configuration library.

Defines

`NET_HOSTNAME_MAX_LEN`

Functions

static inline const char *`net_hostname_get`(void)

Get the device hostname.

Return pointer to device hostname.

Returns Pointer to hostname or NULL if not set.

static inline void `net_hostname_init`(void)

Initialize and set the device hostname.

static inline int `net_hostname_set_postfix`(const uint8_t *hostname_postfix, int postfix_len)

Set the device hostname postfix.

Set the device hostname to some value. This is only used if CONFIG_NET_HOSTNAME_UNIQUE is set.

Parameters

- `hostname_postfix` – Usually link address. The function will convert this to a string.
- `postfix_len` – Length of the hostname_postfix array.

Returns 0 if ok, <0 if error

Network Core Helpers

- *API Reference*

Overview The network subsystem contains two functions for sending and receiving data from the network. The `net_recv_data()` is typically used by network device driver when the received network data needs to be pushed up in the network stack for further processing. All the data is received via a network interface which is typically created by the device driver.

For sending, the `net_send_data()` can be used. Typically applications do not call this function directly as there is the *BSD Sockets* API for sending and receiving network data.

API Reference

group `net_core`

Network core library.

Enums

enum `net_verdict`

Net Verdict.

Values:

enumerator `NET_OK`

Packet has been taken care of.

enumerator `NET_CONTINUE`

Packet has not been touched, other part should decide about its fate.

enumerator `NET_DROP`

Packet must be dropped.

Functions

int `net_recv_data`(struct *net_if* *iface, struct *net_pkt* *pkt)

Called by lower network stack or network device driver when a network packet has been received. The function will push the packet up in the network stack for further processing.

Parameters

- `iface` – Network interface where the packet was received.
- `pkt` – Network packet data.

Returns 0 if ok, <0 if error.

int `net_send_data`(struct *net_pkt* *pkt)

Send data to network.

Send data to network. This should not be used normally by applications as it requires that the network packet is properly constructed.

Parameters

- `pkt` – Network packet.

Returns 0 if ok, <0 if error. If <0 is returned, then the caller needs to unref the pkt in order to avoid memory leak.

Network Interface

Overview The network interface is a nexus that ties the network device drivers and the upper part of the network stack together. All the sent and received data is transferred via a network interface. The network interfaces cannot be created at runtime. A special linker section will contain information about them and that section is populated at linking time.

Network interfaces are created by `NET_DEVICE_INIT()` macro. For Ethernet network, a macro called `ETH_NET_DEVICE_INIT()` should be used instead as it will create VLAN interfaces automatically if `CONFIG_NET_VLAN` is enabled. These macros are typically used in network device driver source code.

The network interface can be turned ON by calling `net_if_up()` and OFF by calling `net_if_down()`. When the device is powered ON, the network interface is also turned ON by default.

The network interfaces can be referenced either by a `struct net_if *` pointer or by a network interface index. The network interface can be resolved from its index by calling `net_if_get_by_index()` and from interface pointer by calling `net_if_get_by_iface()`.

The IP address for network devices must be set for them to be connectable. In a typical dynamic network environment, IP addresses are set automatically by DHCPv4, for example. If needed though, the application can set a device's IP address manually. See the API documentation below for functions such as `net_if_ipv4_addr_add()` that do that.

The `net_if_get_default()` returns a *default* network interface. What this default interface means can be configured via options like `CONFIG_NET_DEFAULT_IF_FIRST` and `CONFIG_NET_DEFAULT_IF_ETHERNET`. See Kconfig file subsys/net/ip/Kconfig what options are available for selecting the default network interface.

The transmitted and received network packets can be classified via a network packet priority. This is typically done in Ethernet networks when virtual LANs (VLANs) are used. Higher priority packets can be sent or received earlier than lower priority packets. The traffic class setup can be configured by `CONFIG_NET_TC_TX_COUNT` and `CONFIG_NET_TC_RX_COUNT` options.

If the `CONFIG_NET_PROMISCUOUS_MODE` is enabled and if the underlaying network technology supports promiscuous mode, then it is possible to receive all the network packets that the network device driver is able to receive. See *Promiscuous Mode* API for more details.

API Reference

group `net_if`

Network Interface abstraction layer.

Defines

`NET_DEVICE_INIT(dev_name, drv_name, init_fn, pm_control_fn, data, cfg, prio, api, l2, l2_ctx_type, mtu)`

Create a network interface and bind it to network device.

Parameters

- dev_name – Network device name.
- drv_name – The name this instance of the driver exposes to the system.
- init_fn – Address to the init function of the driver.
- pm_control_fn – Pointer to pm_control function. Can be NULL if not implemented.
- data – Pointer to the device's private data.
- cfg – The address to the structure containing the configuration information for this instance of the driver.
- prio – The initialization level at which configuration occurs.
- api – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- l2 – Network L2 layer for this network interface.
- l2_ctx_type – Type of L2 context data.
- mtu – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_DEFINE(node_id, init_fn, pm_control_fn, data, cfg, prio, api, l2, l2_ctx_type, mtu)

Like NET_DEVICE_INIT but taking metadata from a devicetree node. Create a network interface and bind it to network device.

Parameters

- node_id – The devicetree node identifier.
- init_fn – Address to the init function of the driver.
- pm_control_fn – Pointer to pm_control function. Can be NULL if not implemented.
- data – Pointer to the device's private data.
- cfg – The address to the structure containing the configuration information for this instance of the driver.
- prio – The initialization level at which configuration occurs.
- api – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- l2 – Network L2 layer for this network interface.
- l2_ctx_type – Type of L2 context data.
- mtu – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_INST_DEFINE(inst, ...)

Like NET_DEVICE_DT_DEFINE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- inst – instance number. This is replaced by DT_DRV_COMPAT(inst) in the call to NET_DEVICE_DT_DEFINE.
- ... – other parameters as expected by NET_DEVICE_DT_DEFINE.

NET_DEVICE_INIT_INSTANCE(dev_name, drv_name, instance, init_fn, pm_control_fn, data, cfg, prio, api, l2, l2_ctx_type, mtu)

Create multiple network interfaces and bind them to network device. If your network device needs more than one instance of a network interface, use this macro below and provide a different instance suffix each time (0, 1, 2, ... or a, b, c ... whatever works for you)

Parameters

- dev_name – Network device name.
- drv_name – The name this instance of the driver exposes to the system.
- instance – Instance identifier.
- init_fn – Address to the init function of the driver.
- pm_control_fn – Pointer to pm_control function. Can be NULL if not implemented.
- data – Pointer to the device's private data.
- cfg – The address to the structure containing the configuration information for this instance of the driver.
- prio – The initialization level at which configuration occurs.
- api – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- l2 – Network L2 layer for this network interface.
- l2_ctx_type – Type of L2 context data.
- mtu – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_DEFINE_INSTANCE(node_id, instance, init_fn, pm_control_fn, data, cfg, prio, api, l2, l2_ctx_type, mtu)

Like NET_DEVICE_OFFLOAD_INIT but taking metadata from a devicetree. Create multiple network interfaces and bind them to network device. If your network device needs more than one instance of a network interface, use this macro below and provide a different instance suffix each time (0, 1, 2, ... or a, b, c ... whatever works for you)

Parameters

- node_id – The devicetree node identifier.
- instance – Instance identifier.
- init_fn – Address to the init function of the driver.
- pm_control_fn – Pointer to pm_control function. Can be NULL if not implemented.
- data – Pointer to the device's private data.
- cfg – The address to the structure containing the configuration information for this instance of the driver.
- prio – The initialization level at which configuration occurs.
- api – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- l2 – Network L2 layer for this network interface.
- l2_ctx_type – Type of L2 context data.
- mtu – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_INST_DEFINE_INSTANCE(inst, ...)

Like NET_DEVICE_DT_DEFINE_INSTANCE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- inst – instance number. This is replaced by DT_DRV_COMPAT(inst) in the call to NET_DEVICE_DT_DEFINE_INSTANCE.
- ... – other parameters as expected by NET_DEVICE_DT_DEFINE_INSTANCE.

NET_DEVICE_OFFLOAD_INIT(dev_name, drv_name, init_fn, pm_control_fn, data, cfg, prio, api, mtu)

Create a offloaded network interface and bind it to network device. The offloaded network interface is implemented by a device vendor HAL or similar.

Parameters

- dev_name – Network device name.
- drv_name – The name this instance of the driver exposes to the system.
- init_fn – Address to the init function of the driver.
- pm_control_fn – Pointer to pm_control function. Can be NULL if not implemented.
- data – Pointer to the device's private data.
- cfg – The address to the structure containing the configuration information for this instance of the driver.
- prio – The initialization level at which configuration occurs.
- api – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- mtu – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_OFFLOAD_DEFINE(node_id, init_fn, pm_control_fn, data, cfg, prio, api, mtu)

Like NET_DEVICE_OFFLOAD_INIT but taking metadata from a devicetree node. Create a offloaded network interface and bind it to network device. The offloaded network interface is implemented by a device vendor HAL or similar.

Parameters

- node_id – The devicetree node identifier.
- init_fn – Address to the init function of the driver.
- pm_control_fn – Pointer to pm_control function. Can be NULL if not implemented.
- data – Pointer to the device's private data.
- cfg – The address to the structure containing the configuration information for this instance of the driver.
- prio – The initialization level at which configuration occurs.
- api – Provides an initial pointer to the API function struct used by the driver. Can be NULL.
- mtu – Maximum transfer unit in bytes for this network interface.

NET_DEVICE_DT_INST_OFFLOAD_DEFINE(inst, ...)

Like NET_DEVICE_DT_OFFLOAD_DEFINE for an instance of a DT_DRV_COMPAT compatible.

Parameters

- inst – instance number. This is replaced by DT_DRV_COMPAT(inst) in the call to NET_DEVICE_DT_OFFLOAD_DEFINE.
- ... – other parameters as expected by NET_DEVICE_DT_OFFLOAD_DEFINE.

Typedefs

typedef void (*net_if_mcast_callback_t)(struct *net_if* *iface, const struct *in6_addr* *addr, bool is_joined)

Define callback that is called whenever IPv6 multicast address group is joined or left.

Param iface A pointer to a struct *net_if* to which the multicast address is attached.

Param addr IPv6 multicast address.

Param is_joined True if the address is joined, false if left.

typedef void (*net_if_link_callback_t)(struct *net_if* *iface, struct *net_linkaddr* *dst, int status)

Define callback that is called after a network packet has been sent.

Param iface A pointer to a struct *net_if* to which the the *net_pkt* was sent to.

Param dst Link layer address of the destination where the network packet was sent.

Param status Send status, 0 is ok, < 0 error.

typedef void (*net_if_cb_t)(struct *net_if* *iface, void *user_data)

Callback used while iterating over network interfaces.

Param iface Pointer to current network interface

Param user_data A valid pointer to user data or NULL

Enums

enum net_if_flag

Values:

enumerator NET_IF_UP

Interface is up/ready to receive and transmit

enumerator NET_IF_POINTOPOINT

Interface is pointopoint

enumerator NET_IF_PROMISC

Interface is in promiscuous mode

enumerator NET_IF_NO_AUTO_START

Do not start the interface immediately after initialization. This requires that either the device driver or some other entity will need to manually take the interface up when needed. For example for Ethernet this will happen when the driver calls the *net_eth_carrier_on()* function.

enumerator NET_IF_SUSPENDED

Power management specific: interface is being suspended

enumerator NET_IF_FORWARD_MULTICASTS

Flag defines if received multicasts of other interface are forwarded on this interface. This activates multicast routing / forwarding for this interface.

enumerator NET_IF_IPV4

Interface supports IPv4

enumerator NET_IF_IPV6

Interface supports IPv6

Functions

static inline void net_if_flag_set(struct *net_if* *iface, enum *net_if_flag* value)

Set a value in network interface flags.

Parameters

- iface – Pointer to network interface
- value – Flag value

static inline bool net_if_flag_test_and_set(struct *net_if* *iface, enum *net_if_flag* value)

Test and set a value in network interface flags.

Parameters

- iface – Pointer to network interface
- value – Flag value

Returns true if the bit was set, false if it wasn't.

static inline void net_if_flag_clear(struct *net_if* *iface, enum *net_if_flag* value)

Clear a value in network interface flags.

Parameters

- iface – Pointer to network interface
- value – Flag value

static inline bool net_if_flag_is_set(struct *net_if* *iface, enum *net_if_flag* value)

Check if a value in network interface flags is set.

Parameters

- iface – Pointer to network interface
- value – Flag value

Returns True if the value is set, false otherwise

enum *net_verdict* net_if_send_data(struct *net_if* *iface, struct *net_pkt* *pkt)

Send a packet through a net iface.

return verdict about the packet

Parameters

- iface – Pointer to a network interface structure
- pkt – Pointer to a net packet to send

static inline const struct *net_l2* *const net_if_l2(struct *net_if* *iface)

Get a pointer to the interface L2.

Parameters

- iface – a valid pointer to a network interface structure

Returns a pointer to the iface L2

enum *net_verdict* `net_if_recv_data`(struct *net_if* *iface, struct *net_pkt* *pkt)

Input a packet through a net iface.

Parameters

- `iface` – Pointer to a network interface structure
- `pkt` – Pointer to a net packet to input

Returns verdict about the packet

static inline void *`net_if_l2_data`(struct *net_if* *iface)

Get a pointer to the interface L2 private data.

Parameters

- `iface` – a valid pointer to a network interface structure

Returns a pointer to the iface L2 data

static inline const struct *device* *`net_if_get_device`(struct *net_if* *iface)

Get an network interface's device.

Parameters

- `iface` – Pointer to a network interface structure

Returns a pointer to the device driver instance

void `net_if_queue_tx`(struct *net_if* *iface, struct *net_pkt* *pkt)

Queue a packet to the net interface TX queue.

Parameters

- `iface` – Pointer to a network interface structure
- `pkt` – Pointer to a net packet to queue

static inline bool `net_if_is_ip_offloaded`(struct *net_if* *iface)

Return the IP offload status.

Parameters

- `iface` – Network interface

Returns True if IP offlining is active, false otherwise.

static inline struct net_offload *`net_if_offload`(struct *net_if* *iface)

Return the IP offload plugin.

Parameters

- `iface` – Network interface

Returns NULL if there is no offload plugin defined, valid pointer otherwise

static inline bool `net_if_is_socket_offloaded`(struct *net_if* *iface)

Return the socket offload status.

Parameters

- `iface` – Network interface

Returns True if socket offloading is active, false otherwise.

static inline struct *net_linkaddr* *`net_if_get_link_addr`(struct *net_if* *iface)

Get an network interface's link address.

Parameters

- `iface` – Pointer to a network interface structure

Returns a pointer to the network link address

static inline struct *net_if_config* *`net_if_get_config`(struct *net_if* *iface)

Return network configuration for this network interface.

Parameters

- `iface` – Pointer to a network interface structure

Returns Pointer to configuration

static inline void `net_if_start_dad`(struct *net_if* *iface)

Start duplicate address detection procedure.

Parameters

- `iface` – Pointer to a network interface structure

void `net_if_start_rs`(struct *net_if* *iface)

Start neighbor discovery and send router solicitation message.

Parameters

- `iface` – Pointer to a network interface structure

static inline void `net_if_stop_rs`(struct *net_if* *iface)

Stop neighbor discovery.

Parameters

- `iface` – Pointer to a network interface structure

static inline int `net_if_set_link_addr`(struct *net_if* *iface, uint8_t *addr, uint8_t len, enum *net_link_type* type)

Set a network interface's link address.

Parameters

- `iface` – Pointer to a network interface structure
- `addr` – A pointer to a uint8_t buffer representing the address. The buffer must remain valid throughout interface lifetime.
- `len` – length of the address buffer
- `type` – network bearer type of this link address

Returns 0 on success

static inline uint16_t `net_if_get_mtu`(struct *net_if* *iface)

Get an network interface's MTU.

Parameters

- `iface` – Pointer to a network interface structure

Returns the MTU

static inline void `net_if_set_mtu`(struct *net_if* *iface, uint16_t mtu)

Set an network interface's MTU.

Parameters

- `iface` – Pointer to a network interface structure
- `mtu` – New MTU, note that we store only 16 bit mtu value.

static inline void net_if_addr_set_lf(struct *net_if_addr* *ifaddr, bool is_infinite)

Set the infinite status of the network interface address.

Parameters

- ifaddr – IP address for network interface
- is_infinite – Infinite status

struct *net_if* *net_if_get_by_link_addr(struct *net_linkaddr* *ll_addr)

Get an interface according to link layer address.

Parameters

- ll_addr – Link layer address.

Returns Network interface or NULL if not found.

struct *net_if* *net_if_lookup_by_dev(const struct *device* *dev)

Find an interface from it's related device.

Parameters

- dev – A valid struct device pointer to relate with an interface

Returns a valid struct *net_if* pointer on success, NULL otherwise

static inline struct *net_if_config* *net_if_config_get(struct *net_if* *iface)

Get network interface IP config.

Parameters

- iface – Interface to use.

Returns NULL if not found or pointer to correct config settings.

void net_if_router_rm(struct *net_if_router* *router)

Remove a router from the system.

Parameters

- router – Pointer to existing router

struct *net_if* *net_if_get_default(void)

Get the default network interface.

Returns Default interface or NULL if no interfaces are configured.

struct *net_if* *net_if_get_first_by_type(const struct *net_l2* *l2)

Get the first network interface according to its type.

Parameters

- l2 – Layer 2 type of the network interface.

Returns First network interface of a given type or NULL if no such interfaces was found.

int net_if_config_ipv6_get(struct *net_if* *iface, struct *net_if_ipv6* **ipv6)

Allocate network interface IPv6 config.

This function will allocate new IPv6 config.

Parameters

- iface – Interface to use.
- ipv6 – Pointer to allocated IPv6 struct is returned to caller.

Returns 0 if ok, <0 if error

int `net_if_config_ipv6_put`(struct *net_if* *iface)

Release network interface IPv6 config.

Parameters

- `iface` – Interface to use.

Returns 0 if ok, <0 if error

struct *net_if_addr* *`net_if_ipv6_addr_lookup`(const struct *in6_addr* *addr, struct *net_if* **iface)

Check if this IPv6 address belongs to one of the interfaces.

Parameters

- `addr` – IPv6 address
- `iface` – Pointer to interface is returned

Returns Pointer to interface address, NULL if not found.

struct *net_if_addr* *`net_if_ipv6_addr_lookup_by_iface`(struct *net_if* *iface, struct *in6_addr* *addr)

Check if this IPv6 address belongs to this specific interfaces.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address

Returns Pointer to interface address, NULL if not found.

int `net_if_ipv6_addr_lookup_by_index`(const struct *in6_addr* *addr)

Check if this IPv6 address belongs to one of the interface indices.

Parameters

- `addr` – IPv6 address

Returns >0 if address was found in given network interface index, all other values mean address was not found

struct *net_if_addr* *`net_if_ipv6_addr_add`(struct *net_if* *iface, struct *in6_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv6 address to an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address
- `addr_type` – IPv6 address type
- `vlifetime` – Validity time for this address

Returns Pointer to interface address, NULL if cannot be added

bool `net_if_ipv6_addr_add_by_index`(int index, struct *in6_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv6 address to an interface by index.

Parameters

- `index` – Network interface index
- `addr` – IPv6 address
- `addr_type` – IPv6 address type
- `vlifetime` – Validity time for this address

Returns True if ok, false if address could not be added

void `net_if_ipv6_addr_update_lifetime`(struct *net_if_addr* *ifaddr, uint32_t vlifetime)

Update validity lifetime time of an IPv6 address.

Parameters

- `ifaddr` – Network IPv6 address
- `vlifetime` – Validity time for this address

bool `net_if_ipv6_addr_rm`(struct *net_if* *iface, const struct *in6_addr* *addr)

Remove an IPv6 address from an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address

Returns True if successfully removed, false otherwise

bool `net_if_ipv6_addr_rm_by_index`(int index, const struct *in6_addr* *addr)

Remove an IPv6 address from an interface by index.

Parameters

- `index` – Network interface index
- `addr` – IPv6 address

Returns True if successfully removed, false otherwise

struct *net_if_mcast_addr* *`net_if_ipv6_maddr_add`(struct *net_if* *iface, const struct *in6_addr* *addr)

Add a IPv6 multicast address to an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 multicast address

Returns Pointer to interface multicast address, NULL if cannot be added

bool `net_if_ipv6_maddr_rm`(struct *net_if* *iface, const struct *in6_addr* *addr)

Remove an IPv6 multicast address from an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 multicast address

Returns True if successfully removed, false otherwise

struct *net_if_mcast_addr* *`net_if_ipv6_maddr_lookup`(const struct *in6_addr* *addr, struct *net_if* **iface)

Check if this IPv6 multicast address belongs to a specific interface or one of the interfaces.

Parameters

- `addr` – IPv6 address
- `iface` – If *iface is null, then pointer to interface is returned, otherwise the *iface value needs to be matched.

Returns Pointer to interface multicast address, NULL if not found.

void net_if_mcast_mon_register(struct *net_if_mcast_monitor* *mon, struct *net_if* *iface, *net_if_mcast_callback_t* cb)

Register a multicast monitor.

Parameters

- mon – Monitor handle. This is a pointer to a monitor storage structure which should be allocated by caller, but does not need to be initialized.
- iface – Network interface
- cb – Monitor callback

void net_if_mcast_mon_unregister(struct *net_if_mcast_monitor* *mon)

Unregister a multicast monitor.

Parameters

- mon – Monitor handle

void net_if_mcast_monitor(struct *net_if* *iface, const struct *in6_addr* *addr, bool is_joined)

Call registered multicast monitors.

Parameters

- iface – Network interface
- addr – Multicast address
- is_joined – Is this multicast address joined (true) or not (false)

void net_if_ipv6_maddr_join(struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be joined.

Parameters

- addr – IPv6 multicast address

static inline bool net_if_ipv6_maddr_is_joined(struct *net_if_mcast_addr* *addr)

Check if given multicast address is joined or not.

Parameters

- addr – IPv6 multicast address

Returns True if address is joined, False otherwise.

void net_if_ipv6_maddr_leave(struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be left.

Parameters

- addr – IPv6 multicast address

struct *net_if_ipv6_prefix* *net_if_ipv6_prefix_get(struct *net_if* *iface, struct *in6_addr* *addr)

Return prefix that corresponds to this IPv6 address.

Parameters

- iface – Network interface
- addr – IPv6 address

Returns Pointer to prefix, NULL if not found.

struct *net_if_ipv6_prefix* *net_if_ipv6_prefix_lookup(struct *net_if* *iface, struct *in6_addr* *addr, uint8_t len)

Check if this IPv6 prefix belongs to this interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 address
- `len` – Prefix length

Returns Pointer to prefix, NULL if not found.

struct *net_if_ipv6_prefix* *`net_if_ipv6_prefix_add`(struct *net_if* *iface, struct *in6_addr* *prefix, uint8_t len, uint32_t lifetime)

Add a IPv6 prefix to an network interface.

Parameters

- `iface` – Network interface
- `prefix` – IPv6 address
- `len` – Prefix length
- `lifetime` – Prefix lifetime in seconds

Returns Pointer to prefix, NULL if the prefix was not added.

bool `net_if_ipv6_prefix_rm`(struct *net_if* *iface, struct *in6_addr* *addr, uint8_t len)

Remove an IPv6 prefix from an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv6 prefix address
- `len` – Prefix length

Returns True if successfully removed, false otherwise

static inline void `net_if_ipv6_prefix_set_lf`(struct *net_if_ipv6_prefix* *prefix, bool is_infinite)

Set the infinite status of the prefix.

Parameters

- `prefix` – IPv6 address
- `is_infinite` – Infinite status

void `net_if_ipv6_prefix_set_timer`(struct *net_if_ipv6_prefix* *prefix, uint32_t lifetime)

Set the prefix lifetime timer.

Parameters

- `prefix` – IPv6 address
- `lifetime` – Prefix lifetime in seconds

void `net_if_ipv6_prefix_unset_timer`(struct *net_if_ipv6_prefix* *prefix)

Unset the prefix lifetime timer.

Parameters

- `prefix` – IPv6 address

bool `net_if_ipv6_addr_onlink`(struct *net_if* **iface, struct *in6_addr* *addr)

Check if this IPv6 address is part of the subnet of our network interface.

Parameters

- `iface` – Network interface. This is returned to the caller. The iface can be NULL in which case we check all the interfaces.
- `addr` – IPv6 address

Returns True if address is part of our subnet, false otherwise

static inline struct *in6_addr* *net_if_router_ipv6(struct *net_if_router* *router)

Get the IPv6 address of the given router.

Parameters

- router – a network router

Returns pointer to the IPv6 address, or NULL if none

struct *net_if_router* *net_if_ipv6_router_lookup(struct *net_if* *iface, struct *in6_addr* *addr)

Check if IPv6 address is one of the routers configured in the system.

Parameters

- iface – Network interface
- addr – IPv6 address

Returns Pointer to router information, NULL if cannot be found

struct *net_if_router* *net_if_ipv6_router_find_default(struct *net_if* *iface, struct *in6_addr* *addr)

Find default router for this IPv6 address.

Parameters

- iface – Network interface. This can be NULL in which case we go through all the network interfaces to find a suitable router.
- addr – IPv6 address

Returns Pointer to router information, NULL if cannot be found

void net_if_ipv6_router_update_lifetime(struct *net_if_router* *router, uint16_t lifetime)

Update validity lifetime time of a router.

Parameters

- router – Network IPv6 address
- lifetime – Lifetime of this router.

struct *net_if_router* *net_if_ipv6_router_add(struct *net_if* *iface, struct *in6_addr* *addr, uint16_t router_lifetime)

Add IPv6 router to the system.

Parameters

- iface – Network interface
- addr – IPv6 address
- router_lifetime – Lifetime of the router

Returns Pointer to router information, NULL if could not be added

bool net_if_ipv6_router_rm(struct *net_if_router* *router)

Remove IPv6 router from the system.

Parameters

- router – Router information.

Returns True if successfully removed, false otherwise

uint8_t net_if_ipv6_get_hop_limit(struct *net_if* *iface)

Get IPv6 hop limit specified for a given interface. This is the default value but can be overridden by the user.

Parameters

- `iface` – Network interface

Returns Hop limit

void `net_ipv6_set_hop_limit`(struct *net_if* *iface, uint8_t hop_limit)

Set the default IPv6 hop limit of a given interface.

Parameters

- `iface` – Network interface
- `hop_limit` – New hop limit

static inline void `net_if_ipv6_set_base_reachable_time`(struct *net_if* *iface, uint32_t reachable_time)

Set IPv6 reachable time for a given interface.

Parameters

- `iface` – Network interface
- `reachable_time` – New reachable time

static inline uint32_t `net_if_ipv6_get_reachable_time`(struct *net_if* *iface)

Get IPv6 reachable timeout specified for a given interface.

Parameters

- `iface` – Network interface

Returns Reachable timeout

uint32_t `net_if_ipv6_calc_reachable_time`(struct *net_if_ipv6* *ipv6)

Calculate next reachable time value for IPv6 reachable time.

Parameters

- `ipv6` – IPv6 address configuration

Returns Reachable time

static inline void `net_if_ipv6_set_reachable_time`(struct *net_if_ipv6* *ipv6)

Set IPv6 reachable time for a given interface. This requires that base reachable time is set for the interface.

Parameters

- `ipv6` – IPv6 address configuration

static inline void `net_if_ipv6_set_retrans_timer`(struct *net_if* *iface, uint32_t retrans_timer)

Set IPv6 retransmit timer for a given interface.

Parameters

- `iface` – Network interface
- `retrans_timer` – New retransmit timer

static inline uint32_t `net_if_ipv6_get_retrans_timer`(struct *net_if* *iface)

Get IPv6 retransmit timer specified for a given interface.

Parameters

- `iface` – Network interface

Returns Retransmit timer

static inline const struct *in6_addr* *net_if_ipv6_select_src_addr(struct *net_if* *iface, const struct *in6_addr* *dst)

Get a IPv6 source address that should be used when sending network data to destination.

Parameters

- `iface` – Interface that was used when packet was received. If the interface is not known, then NULL can be given.
- `dst` – IPv6 destination address

Returns Pointer to IPv6 address to use, NULL if no IPv6 address could be found.

static inline struct *net_if* *net_if_ipv6_select_src_iface(const struct *in6_addr* *dst)

Get a network interface that should be used when sending IPv6 network data to destination.

Parameters

- `dst` – IPv6 destination address

Returns Pointer to network interface to use, NULL if no suitable interface could be found.

struct *in6_addr* *net_if_ipv6_get_ll(struct *net_if* *iface, enum *net_addr_state* addr_state)

Get a IPv6 link local address in a given state.

Parameters

- `iface` – Interface to use. Must be a valid pointer to an interface.
- `addr_state` – IPv6 address state (preferred, tentative, deprecated)

Returns Pointer to link local IPv6 address, NULL if no proper IPv6 address could be found.

struct *in6_addr* *net_if_ipv6_get_ll_addr(enum *net_addr_state* state, struct *net_if* **iface)

Return link local IPv6 address from the first interface that has a link local address matching give state.

Parameters

- `state` – IPv6 address state (ANY, TENTATIVE, PREFERRED, DEPRECATED)
- `iface` – Pointer to interface is returned

Returns Pointer to IPv6 address, NULL if not found.

void net_if_ipv6_dad_failed(struct *net_if* *iface, const struct *in6_addr* *addr)

Stop IPv6 Duplicate Address Detection (DAD) procedure if we find out that our IPv6 address is already in use.

Parameters

- `iface` – Interface where the DAD was running.
- `addr` – IPv6 address that failed DAD

struct *in6_addr* *net_if_ipv6_get_global_addr(enum *net_addr_state* state, struct *net_if* **iface)

Return global IPv6 address from the first interface that has a global IPv6 address matching the given state.

Parameters

- `state` – IPv6 address state (ANY, TENTATIVE, PREFERRED, DEPRECATED)
- `iface` – Caller can give an interface to check. If iface is set to NULL, then all the interfaces are checked. Pointer to interface where the IPv6 address is defined is returned to the caller.

Returns Pointer to IPv6 address, NULL if not found.

int `net_if_config_ipv4_get`(struct *net_if* *iface, struct *net_if_ipv4* **ipv4)

Allocate network interface IPv4 config.

This function will allocate new IPv4 config.

Parameters

- `iface` – Interface to use.
- `ipv4` – Pointer to allocated IPv4 struct is returned to caller.

Returns 0 if ok, <0 if error

int `net_if_config_ipv4_put`(struct *net_if* *iface)

Release network interface IPv4 config.

Parameters

- `iface` – Interface to use.

Returns 0 if ok, <0 if error

uint8_t `net_if_ipv4_get_ttl`(struct *net_if* *iface)

Get IPv4 time-to-live value specified for a given interface.

Parameters

- `iface` – Network interface

Returns Time-to-live

void `net_if_ipv4_set_ttl`(struct *net_if* *iface, uint8_t ttl)

Set IPv4 time-to-live value specified to a given interface.

Parameters

- `iface` – Network interface
- `ttl` – Time-to-live value

struct *net_if_addr* *`net_if_ipv4_addr_lookup`(const struct *in_addr* *addr, struct *net_if* **iface)

Check if this IPv4 address belongs to one of the interfaces.

Parameters

- `addr` – IPv4 address
- `iface` – Interface is returned

Returns Pointer to interface address, NULL if not found.

struct *net_if_addr* *`net_if_ipv4_addr_add`(struct *net_if* *iface, struct *in_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv4 address to an interface.

Parameters

- `iface` – Network interface
- `addr` – IPv4 address
- `addr_type` – IPv4 address type
- `vlifetime` – Validity time for this address

Returns Pointer to interface address, NULL if cannot be added

bool net_if_ipv4_addr_rm(struct *net_if* *iface, const struct *in_addr* *addr)

Remove a IPv4 address from an interface.

Parameters

- iface – Network interface
- addr – IPv4 address

Returns True if successfully removed, false otherwise

int net_if_ipv4_addr_lookup_by_index(const struct *in_addr* *addr)

Check if this IPv4 address belongs to one of the interface indices.

Parameters

- addr – IPv4 address

Returns >0 if address was found in given network interface index, all other values mean address was not found

bool net_if_ipv4_addr_add_by_index(int index, struct *in_addr* *addr, enum *net_addr_type* addr_type, uint32_t vlifetime)

Add a IPv4 address to an interface by network interface index.

Parameters

- index – Network interface index
- addr – IPv4 address
- addr_type – IPv4 address type
- vlifetime – Validity time for this address

Returns True if ok, false if the address could not be added

bool net_if_ipv4_addr_rm_by_index(int index, const struct *in_addr* *addr)

Remove a IPv4 address from an interface by interface index.

Parameters

- index – Network interface index
- addr – IPv4 address

Returns True if successfully removed, false otherwise

struct *net_if_mcast_addr* *net_if_ipv4_maddr_add(struct *net_if* *iface, const struct *in_addr* *addr)

Add a IPv4 multicast address to an interface.

Parameters

- iface – Network interface
- addr – IPv4 multicast address

Returns Pointer to interface multicast address, NULL if cannot be added

bool net_if_ipv4_maddr_rm(struct *net_if* *iface, const struct *in_addr* *addr)

Remove an IPv4 multicast address from an interface.

Parameters

- iface – Network interface
- addr – IPv4 multicast address

Returns True if successfully removed, false otherwise

struct *net_if_mcast_addr* *net_if_ipv4_maddr_lookup(const struct *in_addr* *addr, struct *net_if* **iface)

Check if this IPv4 multicast address belongs to a specific interface or one of the interfaces.

Parameters

- addr – IPv4 address
- iface – If *iface is null, then pointer to interface is returned, otherwise the *iface value needs to be matched.

Returns Pointer to interface multicast address, NULL if not found.

void net_if_ipv4_maddr_join(struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be joined.

Parameters

- addr – IPv4 multicast address

static inline bool net_if_ipv4_maddr_is_joined(struct *net_if_mcast_addr* *addr)

Check if given multicast address is joined or not.

Parameters

- addr – IPv4 multicast address

Returns True if address is joined, False otherwise.

void net_if_ipv4_maddr_leave(struct *net_if_mcast_addr* *addr)

Mark a given multicast address to be left.

Parameters

- addr – IPv4 multicast address

static inline struct *in_addr* *net_if_router_ipv4(struct *net_if_router* *router)

Get the IPv4 address of the given router.

Parameters

- router – a network router

Returns pointer to the IPv4 address, or NULL if none

struct *net_if_router* *net_if_ipv4_router_lookup(struct *net_if* *iface, struct *in_addr* *addr)

Check if IPv4 address is one of the routers configured in the system.

Parameters

- iface – Network interface
- addr – IPv4 address

Returns Pointer to router information, NULL if cannot be found

struct *net_if_router* *net_if_ipv4_router_find_default(struct *net_if* *iface, struct *in_addr* *addr)

Find default router for this IPv4 address.

Parameters

- iface – Network interface. This can be NULL in which case we go through all the network interfaces to find a suitable router.
- addr – IPv4 address

Returns Pointer to router information, NULL if cannot be found

struct *net_if_router* *net_if_ipv4_router_add(struct *net_if* *iface, struct *in_addr* *addr, bool is_default, uint16_t router_lifetime)

Add IPv4 router to the system.

Parameters

- `iface` – Network interface
- `addr` – IPv4 address
- `is_default` – Is this router the default one
- `router_lifetime` – Lifetime of the router

Returns Pointer to router information, NULL if could not be added

bool net_if_ipv4_router_rm(struct *net_if_router* *router)

Remove IPv4 router from the system.

Parameters

- `router` – Router information.

Returns True if successfully removed, false otherwise

bool net_if_ipv4_addr_mask_cmp(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given IPv4 address belongs to local subnet.

Parameters

- `iface` – Interface to use. Must be a valid pointer to an interface.
- `addr` – IPv4 address

Returns True if address is part of local subnet, false otherwise.

bool net_if_ipv4_is_addr_bcast(struct *net_if* *iface, const struct *in_addr* *addr)

Check if the given IPv4 address is a broadcast address.

Parameters

- `iface` – Interface to use. Must be a valid pointer to an interface.
- `addr` – IPv4 address, this should be in network byte order

Returns True if address is a broadcast address, false otherwise.

static inline struct *net_if* *net_if_ipv4_select_src_iface(const struct *in_addr* *dst)

Get a network interface that should be used when sending IPv4 network data to destination.

Parameters

- `dst` – IPv4 destination address

Returns Pointer to network interface to use, NULL if no suitable interface could be found.

static inline const struct *in_addr* *net_if_ipv4_select_src_addr(struct *net_if* *iface, const struct *in_addr* *dst)

Get a IPv4 source address that should be used when sending network data to destination.

Parameters

- `iface` – Interface to use when sending the packet. If the interface is not known, then NULL can be given.
- `dst` – IPv4 destination address

Returns Pointer to IPv4 address to use, NULL if no IPv4 address could be found.

struct *in_addr* *net_if_ipv4_get_ll(struct *net_if* *iface, enum *net_addr_state* addr_state)

Get a IPv4 link local address in a given state.

Parameters

- iface – Interface to use. Must be a valid pointer to an interface.
- addr_state – IPv4 address state (preferred, tentative, deprecated)

Returns Pointer to link local IPv4 address, NULL if no proper IPv4 address could be found.

struct *in_addr* *net_if_ipv4_get_global_addr(struct *net_if* *iface, enum *net_addr_state* addr_state)

Get a IPv4 global address in a given state.

Parameters

- iface – Interface to use. Must be a valid pointer to an interface.
- addr_state – IPv4 address state (preferred, tentative, deprecated)

Returns Pointer to link local IPv4 address, NULL if no proper IPv4 address could be found.

void net_if_ipv4_set_netmask(struct *net_if* *iface, const struct *in_addr* *netmask)

Set IPv4 netmask for an interface.

Parameters

- iface – Interface to use.
- netmask – IPv4 netmask

bool net_if_ipv4_set_netmask_by_index(int index, const struct *in_addr* *netmask)

Set IPv4 netmask for an interface index.

Parameters

- index – Network interface index
- netmask – IPv4 netmask

Returns True if netmask was added, false otherwise.

void net_if_ipv4_set_gw(struct *net_if* *iface, const struct *in_addr* *gw)

Set IPv4 gateway for an interface.

Parameters

- iface – Interface to use.
- gw – IPv4 address of an gateway

bool net_if_ipv4_set_gw_by_index(int index, const struct *in_addr* *gw)

Set IPv4 gateway for an interface index.

Parameters

- index – Network interface index
- gw – IPv4 address of an gateway

Returns True if gateway was added, false otherwise.

struct *net_if* *net_if_select_src_iface(const struct *sockaddr* *dst)

Get a network interface that should be used when sending IPv6 or IPv4 network data to destination.

Parameters

- `dst` – IPv6 or IPv4 destination address

Returns Pointer to network interface to use. Note that the function will return the default network interface if the best network interface is not found.

void `net_if_register_link_cb`(struct *net_if_link_cb* *link, *net_if_link_callback_t* cb)

Register a link callback.

Parameters

- `link` – Caller specified handler for the callback.
- `cb` – Callback to register.

void `net_if_unregister_link_cb`(struct *net_if_link_cb* *link)

Unregister a link callback.

Parameters

- `link` – Caller specified handler for the callback.

void `net_if_call_link_cb`(struct *net_if* *iface, struct *net_linkaddr* *lladdr, int status)

Call a link callback function.

Parameters

- `iface` – Network interface.
- `lladdr` – Destination link layer address
- `status` – 0 is ok, < 0 error

bool `net_if_need_calc_rx_checksum`(struct *net_if* *iface)

Check if received network packet checksum calculation can be avoided or not. For example many ethernet devices support network packet offloading in which case the IP stack does not need to calculate the checksum.

Parameters

- `iface` – Network interface

Returns True if checksum needs to be calculated, false otherwise.

bool `net_if_need_calc_tx_checksum`(struct *net_if* *iface)

Check if network packet checksum calculation can be avoided or not when sending the packet. For example many ethernet devices support network packet offloading in which case the IP stack does not need to calculate the checksum.

Parameters

- `iface` – Network interface

Returns True if checksum needs to be calculated, false otherwise.

struct *net_if* *`net_if_get_by_index`(int index)

Get interface according to index.

This is a syscall only to provide access to the object for purposes of assigning permissions.

Parameters

- `index` – Interface index

Returns Pointer to interface or NULL if not found.

int `net_if_get_by_iface`(struct *net_if* *iface)

Get interface index according to pointer.

Parameters

- `iface` – Pointer to network interface

Returns Interface index

void `net_if_foreach`(*net_if_cb_t* cb, void *user_data)

Go through all the network interfaces and call callback for each interface.

Parameters

- `cb` – User-supplied callback function to call
- `user_data` – User specified data

int `net_if_up`(struct *net_if* *iface)

Bring interface up.

Parameters

- `iface` – Pointer to network interface

Returns 0 on success

static inline bool `net_if_is_up`(struct *net_if* *iface)

Check if interface is up.

Parameters

- `iface` – Pointer to network interface

Returns True if interface is up, False if it is down.

int `net_if_down`(struct *net_if* *iface)

Bring interface down.

Parameters

- `iface` – Pointer to network interface

Returns 0 on success

int `net_if_set_promisc`(struct *net_if* *iface)

Set network interface into promiscuous mode.

Note that not all network technologies will support this.

Parameters

- `iface` – Pointer to network interface

Returns 0 on success, <0 if error

void `net_if_unset_promisc`(struct *net_if* *iface)

Set network interface into normal mode.

Parameters

- `iface` – Pointer to network interface

bool `net_if_is_promisc`(struct *net_if* *iface)

Check if promiscuous mode is set or not.

Parameters

- `iface` – Pointer to network interface

Returns True if interface is in promisc mode, False if interface is not in in promiscuous mode.

static inline bool `net_if_are_pending_tx_packets`(struct *net_if* *iface)

Check if there are any pending TX network data for a given network interface.

Parameters

- `iface` – Pointer to network interface

Returns True if there are pending TX network packets for this network interface, False otherwise.

struct `net_if_addr`

#include <net_if.h> Network Interface unicast IP addresses.

Stores the unicast IP addresses assigned to this network interface.

Public Members

struct net_addr `address`

IP address

enum *net_addr_type* `addr_type`

How the IP address was set

enum *net_addr_state* `addr_state`

What is the current state of the address

uint8_t `is_infinite`

Is the IP address valid forever

uint8_t `is_used`

Is this IP address used or not

uint8_t `is_mesh_local`

Is this IP address usage limited to the subnet (mesh) or not

struct `net_if_mcast_addr`

#include <net_if.h> Network Interface multicast IP addresses.

Stores the multicast IP addresses assigned to this network interface.

Public Members

struct net_addr `address`

IP address

uint8_t `is_used`

Is this multicast IP address used or not

uint8_t `is_joined`

Did we join to this group

struct `net_if_ipv6_prefix`

#include <net_if.h> Network Interface IPv6 prefixes.

Stores the multicast IP addresses assigned to this network interface.

Public Members

struct *net_timeout* `lifetime`
Prefix lifetime

struct *in6_addr* `prefix`
IPv6 prefix

struct *net_if* *`iface`
Backpointer to network interface where this prefix is used

uint8_t `len`
Prefix length

uint8_t `is_infinite`
Is the IP prefix valid forever

uint8_t `is_used`
Is this prefix used or not

struct `net_if_router`
#include <net_if.h> Information about routers in the system.

Stores the router information.

Public Members

sys_snode_t `node`
Slist lifetime timer node

struct net_addr `address`
IP address

struct *net_if* *`iface`
Network interface the router is connected to

uint32_t `life_start`
Router life timer start

uint16_t `lifetime`
Router lifetime

uint8_t `is_used`
Is this router used or not

uint8_t `is_default`
Is default router

uint8_t `is_infinite`

Is the router valid forever

struct `net_if_ipv6`

#include <net_if.h>

Public Members

struct *net_if_addr* `unicast`[NET_IF_MAX_IPV6_ADDR]

Unicast IP addresses

struct *net_if_mcast_addr* `mcast`[NET_IF_MAX_IPV6_MADDR]

Multicast IP addresses

struct *net_if_ipv6_prefix* `prefix`[NET_IF_MAX_IPV6_PREFIX]

Prefixes

uint32_t `base_reachable_time`

Default reachable time (RFC 4861, page 52)

uint32_t `reachable_time`

Reachable time (RFC 4861, page 20)

uint32_t `retrans_timer`

Retransmit timer (RFC 4861, page 52)

uint8_t `hop_limit`

IPv6 hop limit

struct `net_if_ipv4`

#include <net_if.h>

Public Members

struct *net_if_addr* `unicast`[NET_IF_MAX_IPV4_ADDR]

Unicast IP addresses

struct *net_if_mcast_addr* `mcast`[NET_IF_MAX_IPV4_MADDR]

Multicast IP addresses

struct *in_addr* `gw`

Gateway

struct *in_addr* `netmask`

Netmask

uint8_t `ttl`

IPv4 time-to-live

struct `net_if_ip`

#include <net_if.h> Network interface IP address configuration.

struct `net_if_config`

#include <net_if.h> IP and other configuration related data for network interface.

Public Members

struct *net_if_ip* `ip`

IP address configuration setting

struct `net_traffic_class`

#include <net_if.h> Network traffic class.

Traffic classes are used when sending or receiving data that is classified with different priorities. So some traffic can be marked as high priority and it will be sent or received first. Each network packet that is transmitted or received goes through a fifo to a thread that will transmit it.

Public Members

struct k_fifo `fifo`

Fifo for handling this Tx or Rx packet

struct *k_thread* `handler`

Traffic class handler thread

k_thread_stack_t *`stack`

Stack for this handler

struct `net_if_dev`

#include <net_if.h> Network Interface Device structure.

Used to handle a network interface on top of a device driver instance. There can be many *net_if_dev* instance against the same device.

Such interface is mainly to be used by the link layer, but is also tight to a network context: it then makes the relation with a network context and the network device.

Because of the strong relationship between a device driver and such network interface, each *net_if_dev* should be instantiated by

Public Members

const struct *device* *`dev`

The actually device driver instance the *net_if* is related to

const struct *net_l2* *const `l2`

Interface's L2 layer

void *`l2_data`

Interface's private L2 data pointer

struct *net_linkaddr* `link_addr`

The hardware link address

uint16_t `mtu`

The hardware MTU

struct `net_if`

#include <net_if.h> Network Interface structure.

Used to handle a network interface on top of a *net_if_dev* instance. There can be many *net_if* instance against the same *net_if_dev* instance.

Public Members

struct *net_if_dev* *`if_dev`

The *net_if_dev* instance the *net_if* is related to

struct *net_if_config* `config`

Network interface instance configuration

struct `net_if_mcast_monitor`

#include <net_if.h> Multicast monitor handler struct.

Stores the multicast callback information. Caller must make sure that the variable pointed by this is valid during the lifetime of registration. Typically this means that the variable cannot be allocated from stack.

Public Members

sys_snode_t `node`

Node information for the slist.

struct *net_if* *`iface`

Network interface

net_if_mcast_callback_t `cb`

Multicast callback

struct `net_if_link_cb`

#include <net_if.h> Link callback handler struct.

Stores the link callback information. Caller must make sure that the variable pointed by this is valid during the lifetime of registration. Typically this means that the variable cannot be allocated from stack.

Public Members

sys_snode_t `node`

Node information for the slist.

net_if_link_callback_t `cb`

Link callback

L2 Layer Management

Overview The L2 stack is designed to hide the whole networking link-layer part and the related device drivers from the upper network stack. This is made through a *net_if* declared in include/net/net_if.h.

The upper layers are unaware of implementation details beyond the net_if object and the generic API provided by the L2 layer in include/net/net_l2.h as *net_l2*.

Only the L2 layer can talk to the device driver, linked to the net_if object. The L2 layer dictates the API provided by the device driver, specific for that device, and optimized for working together.

Currently, there are L2 layers for *Ethernet*, *IEEE 802.15.4 Soft-MAC*, Bluetooth IPSP, *CANBUS*, *OpenThread*, Wi-Fi, and a dummy layer example that can be used as a template for writing a new one.

L2 layer API In order to create an L2 layer, or a driver for a specific L2 layer, one needs to understand how the L3 layer interacts with it and how the L2 layer is supposed to behave. See also *network stack architecture* for more details. The generic L2 API has these functions:

- `recv()`: All device drivers, once they receive a packet which they put into a *net_pkt*, will push this buffer to the network stack via *net_recv_data()*. At this point, the network stack does not know what to do with it. Instead, it passes the buffer along to the L2 stack's `recv()` function for handling. The L2 stack does what it needs to do with the packet, for example, parsing the link layer header, or handling link-layer only packets. The `recv()` function will return `NET_DROP` in case of an erroneous packet, `NET_OK` if the packet was fully consumed by the L2, or `NET_CONTINUE` if the network stack should then handle it.
- `send()`: Similar to receive function, the network stack will call this function to actually send a network packet. All relevant link-layer content will be generated and added by this function. The `send()` function returns the number of bytes sent, or a negative error code if there was a failure sending the network packet.
- `enable()`: This function is used to enable/disable traffic over a network interface. The function returns `<0` if error and `>=0` if no error.
- `get_flags()`: This function will return the capabilities of an L2 driver, for example whether the L2 supports multicast or promiscuous mode.

Network Device drivers Network device drivers fully follows Zephyr device driver model as a basis. Please refer to *Device Driver Model*.

There are, however, two differences:

- The driver_api pointer must point to a valid `net_if_api` pointer.
- The network device driver must use *`NET_DEVICE_INIT_INSTANCE()`* or *`ETH_NET_DEVICE_INIT()`* for Ethernet devices. These macros will call the *`DEVICE_DEFINE()`* macro, and also instantiate a unique *`net_if`* related to the created device driver instance.

Implementing a network device driver depends on the L2 stack it belongs to: *Ethernet*, *IEEE 802.15.4*, etc. In the next section, we will describe how a device driver should behave when receiving or sending a network packet. The rest is hardware dependent and is not detailed here.

Ethernet device driver On reception, it is up to the device driver to fill-in the network packet with as many data buffers as required. The network packet itself is a *`net_pkt`* and should be allocated through *`net_pkt_rx_alloc_with_buffer()`*. Then all data buffers will be automatically allocated and filled by *`net_pkt_write()`*.

After all the network data has been received, the device driver needs to call *`net_recv_data()`*. If that call fails, it will be up to the device driver to unreference the buffer via *`net_pkt_unref()`*.

On sending, the device driver send function will be called, and it is up to the device driver to send the network packet all at once, with all the buffers.

Each Ethernet device driver will need, in the end, to call `ETH_NET_DEVICE_INIT()` like this:

```
ETH_NET_DEVICE_INIT(..., CONFIG_ETH_INIT_PRIORITY,
                    &the_valid_net_if_api_instance, 1500);
```

IEEE 802.15.4 device driver Device drivers for IEEE 802.15.4 L2 work basically the same as for Ethernet. What has been described above, especially for `recv()`, applies here as well. There are two specific differences however:

- It requires a dedicated device driver API: *`ieee802154_radio_api`*, which overloads `net_if_api`. This is because 802.15.4 L2 needs more from the device driver than just `send()` and `recv()` functions. This dedicated API is declared in include/net/ieee802154_radio.h. Each and every IEEE 802.15.4 device driver must provide a valid pointer on such relevantly filled-in API structure.
- Sending a packet is slightly different than in Ethernet. IEEE 802.15.4 sends relatively small frames, 127 bytes all inclusive: frame header, payload and frame checksum. Buffers are meant to fit such frame size limitation. But a buffer containing an IPv6/UDP packet might have more than one fragment. IEEE 802.15.4 drivers handle only one buffer at a time. This is why the *`ieee802154_radio_api`* requires a tx function pointer which differs from the `net_if_api` send function pointer. Instead, the IEEE 802.15.4 L2, provides a generic `ieee802154_radio_send()` meant to be given as *`net_if`* send function. It turn, the implementation of `ieee802154_radio_send()` will ensure the same behavior: sending one buffer at a time through *`ieee802154_radio_api`* tx function, and unreferencing the network packet only when all the transmission were successful.

Each IEEE 802.15.4 device driver, in the end, will need to call `NET_DEVICE_INIT_INSTANCE()` that way:

```
NET_DEVICE_INIT_INSTANCE(...,
                         the_device_init_prio,
                         &the_valid_ieee802154_radio_api_instance,
                         IEEE802154_L2,
                         NET_L2_GET_CTX_TYPE(IEEE802154_L2), 125);
```

API Reference

group net_l2

Network Layer 2 abstraction layer.

Enums

enum net_l2_flags

L2 flags

Values:

enumerator NET_L2_MULTICAST = *BIT*(0)

IP multicast supported

enumerator NET_L2_MULTICAST_SKIP_JOIN_SOLICIT_NODE = *BIT*(1)

Do not joint solicited node multicast group

enumerator NET_L2_PROMISC_MODE = *BIT*(2)

Is promiscuous mode supported

enumerator NET_L2_POINT_TO_POINT = *BIT*(3)

Is this L2 point-to-point with tunneling so no need to have IP address etc to network interface.

struct net_l2

#include <net_l2.h> Network L2 structure.

Used to provide an interface to lower network stack.

Public Members

enum *net_verdict* (*recv)(struct *net_if* *iface, struct *net_pkt* *pkt)

This function is used by net core to get iface's L2 layer parsing what's relevant to itself.

int (*send)(struct *net_if* *iface, struct *net_pkt* *pkt)

This function is used by net core to push a packet to lower layer (interface's L2), which in turn might work on the packet relevantly. (adding proper header etc...) Returns a negative error code, or the number of bytes sent otherwise.

int (*enable)(struct *net_if* *iface, bool state)

This function is used to enable/disable traffic over a network interface. The function returns <0 if error and >=0 if no error.

enum *net_l2_flags* (*get_flags)(struct *net_if* *iface)

Return L2 flags for the network interface.

Network Traffic Offloading

Network Offloading

Overview The network offloading API provides hooks that a device vendor can use to provide an alternate implementation for an IP stack. This means that the actual network connection creation, data transfer, etc., is done in the vendor HAL instead of the Zephyr network stack.

API Reference

group `net_offload`

Network offloading interface.

Socket Offloading

Overview In addition to the network offloading API, Zephyr allows offloading of networking functionality at the socket API level. With this approach, vendors who provide an alternate implementation of the networking stack, exposing socket API for their networking devices, can easily integrate it with Zephyr.

See drivers/wifi/simplelink/simplelink_sockets.c for a sample implementation on how to integrate network offloading at socket level.

Link Layer Address Handling

Overview The link layer addresses are set for network interfaces so that L2 connectivity works correctly in the network stack. Typically the link layer addresses are 6 bytes long like in Ethernet but for IEEE 802.15.4 the link layer address length is 8 bytes.

API Reference

group `net_linkaddr`

Network link address library.

Defines

NET_LINK_ADDR_MAX_LENGTH

Maximum length of the link address

Enums

enum net_link_type

Type of the link address. This indicates the network technology that this address is used in. Note that in order to save space we store the value into a uint8_t variable, so please do not introduce any values > 255 in this enum.

Values:

enumerator NET_LINK_UNKNOWN = 0

Unknown link address type.

enumerator NET_LINK_IEEE802154

IEEE 802.15.4 link address.

enumerator NET_LINK_BLUETOOTH

Bluetooth IPSP link address.

enumerator NET_LINK_ETHERNET

Ethernet link address.

enumerator NET_LINK_DUMMY

Dummy link address. Used in testing apps and loopback support.

enumerator NET_LINK_CANBUS_RAW

CANBUS link address.

enumerator NET_LINK_CANBUS

6loCAN link address.

Functions

static inline bool net_linkaddr_cmp(struct *net_linkaddr* *lladdr1, struct *net_linkaddr* *lladdr2)

Compare two link layer addresses.

Parameters

- lladdr1 – Pointer to a link layer address
- lladdr2 – Pointer to a link layer address

Returns True if the addresses are the same, false otherwise.

static inline int net_linkaddr_set(struct *net_linkaddr_storage* *lladdr_store, uint8_t *new_addr, uint8_t new_len)

Set the member data of a link layer address storage structure.

Parameters

- `lladdr_store` – The link address storage structure to change.
- `new_addr` – Array of bytes containing the link address.
- `new_len` – Length of the link address array. This value should always be <= NET_LINK_ADDR_MAX_LENGTH.

struct `net_linkaddr`

#include <net_linkaddr.h> Hardware link address structure.

Used to hold the link address information

Public Members

uint8_t *`addr`

The array of byte representing the address

uint8_t `len`

Length of that address array

uint8_t `type`

What kind of address is this for

struct `net_linkaddr_storage`

#include <net_linkaddr.h> Hardware link address structure.

Used to hold the link address information. This variant is needed when we have to store the link layer address.

Note that you cannot cast this to *net_linkaddr* as uint8_t * is handled differently than uint8_t addr[] and the fields are purposely in different order.

Public Members

uint8_t `type`

What kind of address is this for

uint8_t `len`

The real length of the ll address.

uint8_t `addr`[6]

The array of bytes representing the address

Ethernet Management

- *Overview*
- *API Reference*

Overview Ethernet management API provides functions to manage the Ethernet network interface low level status. The caller of these functions can:

- raise `carrier ON` or `carrier OFF` management events
- raise `VLAN enabled` or `VLAN disabled` management events

Typically the `carrier OFF` event would be generated by the Ethernet device driver when it notices that the Ethernet cable is disconnected. The `carrier ON` event would be generated if the Ethernet device driver notices that the Ethernet cable is re-connected.

Currently the VLAN events are generated by the Ethernet L2 layer when a specific VLAN tag is either enabled or disabled.

The user application can monitor these events if it needs to act when the corresponding status changes.

API Reference

group `ethernet_mgmt`

Ethernet library.

Functions

void `ethernet_mgmt_raise_carrier_on_event`(struct *net_if* *iface)

Raise CARRIER_ON event when Ethernet is connected.

Parameters

- `iface` – Ethernet network interface.

void `ethernet_mgmt_raise_carrier_off_event`(struct *net_if* *iface)

Raise CARRIER_OFF event when Ethernet is disconnected.

Parameters

- `iface` – Ethernet network interface.

void `ethernet_mgmt_raise_vlan_enabled_event`(struct *net_if* *iface, uint16_t tag)

Raise VLAN_ENABLED event when VLAN is enabled.

Parameters

- `iface` – Ethernet network interface.
- `tag` – VLAN tag which is enabled.

void `ethernet_mgmt_raise_vlan_disabled_event`(struct *net_if* *iface, uint16_t tag)

Raise VLAN_DISABLED event when VLAN is disabled.

Parameters

- `iface` – Ethernet network interface.
- `tag` – VLAN tag which is disabled.

Traffic Classification

Overview Traffic classification is an automated process that categorizes computer network traffic according to various parameters. For Zephyr, the VLAN priority code point (PCP) is used to classify both received and sent network packets. See more information about VLAN priority at IEEE 802.1Q.

By default, all network traffic is treated equal in Zephyr. If desired, the option `CONFIG_NET_TC_TX_COUNT` can be used to set the number of transmit queues. The option `CONFIG_NET_TC_RX_COUNT` can be used to set the number of receive queues. Each traffic class queue corresponds to a specific kernel work queue.

Each kernel work queue has a priority. The VLAN priority is mapped to a certain traffic class according to rules specified in IEEE 802.1Q spec chapter I.3, chapter 8.6.6 table 8-4, and chapter 34.5 table 34-1. Each traffic class is in turn mapped to a certain kernel work queue. The maximum number of traffic classes for both Rx and Tx is 8.

See subsys/net/ip/net_tc.c for details of how various mappings are done.

Network Shell

Network shell provides helpers for figuring out network status, enabling/disabling features, and issuing commands like ping or DNS resolving. Note that `net-shell` should probably not be used in production code as it will require extra memory. See also *generic shell* for detailed shell information.

The following net-shell commands are implemented:

Table 6: net-shell commands

Command	Description
net allocs	Print network memory allocations. Only available if `CONFIG_NET_DEBUG_NET_PKT_ALLOC` is set.
net arp	Print information about IPv4 ARP cache. Only available if `CONFIG_NET_ARP` is set in IPv4 enabled networks.
net capture	Monitor network traffic See *Monitor Network Traffic* for details.
net conn	Print information about network connections.
net dns	Show how DNS is configured. The command can also be used to resolve a DNS name. Only available if `CONFIG_DNS_RESOLVER` is set.
net events	Enable network event monitoring. Only available if `CONFIG_NET_MGMT_EVENT_MONITOR` is set.
net gptp	Print information about gPTP support. Only available if `CONFIG_NET_GPTP` is set.
net iface	Print information about network interfaces.
net ipv6	Print IPv6 specific information and configuration. Only available if `CONFIG_NET_IPV6` is set.
net mem	Print information about network memory usage. The command will print more information if `CONFIG_NET_BUF_POOL_USAGE` is set.
net nbr	Print neighbor information. Only available if `CONFIG_NET_IPV6` is set.
net ping	Ping a network host.
net route	Show IPv6 network routes. Only available if `CONFIG_NET_ROUTE` is set.
net stats	Show network statistics.
net tcp	Connect/send data/close TCP connection. Only available if `CONFIG_NET_TCP` is set.
net vlan	Show Ethernet virtual LAN information. Only available if `CONFIG_NET_VLAN` is set.

7.20.6 Time Sensitive Networking

generic Precision Time Protocol (gPTP)

Overview This gPTP stack supports the protocol and procedures as defined in the IEEE 802.1AS-2011 standard (Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks).

Supported features The stack handles communications and state machines defined in the IEEE 802.1AS-2011 standard. Mandatory requirements for a full-duplex point-to-point link endpoint, as defined in Annex A of the standard, are supported.

The stack is in principle capable of handling communications on multiple network interfaces (also defined as "ports" in the standard) and thus act as a 802.1AS bridge. However, this mode of operation has not been validated on the Zephyr OS.

Supported hardware Although the stack itself is hardware independent, Ethernet frame timestamping support must be enabled in ethernet drivers.

Boards supported:

- frdm_k64f
- sam_e70_xplained
- native_posix (only usable for simple testing, limited capabilities due to lack of hardware clock)
- qemu_x86 (emulated, limited capabilities due to lack of hardware clock)

Enabling the stack The following configuration option must me enabled in `prj.conf` file.

- `CONFIG_NET_GPTP`

Application interfaces Only two Application Interfaces as defined in section 9 of the standard are available:

- `ClockTargetPhaseDiscontinuity` interface (*gptp_register_phase_dis_cb()*)
- `ClockTargetEventCapture` interface (*gptp_event_capture()*)

Testing The stack has been informally tested using the OpenAVnu gPTP and Linux ptp4l daemons. The gPTP sample application from the Zephyr source distribution can be used for testing.

API Reference

group `gptp`

generic Precision Time Protocol (gPTP) support

Typedefs

typedef void (*gptp_phase_dis_callback_t)(uint8_t *gm_identity, uint16_t *time_base, struct *gptp_scaled_ns* *last_gm_ph_change, double *last_gm_freq_change)

Define callback that is called after a phase discontinuity has been sent by the grandmaster.

Param gm_identity A pointer to first element of a ClockIdentity array. The size of the array is GPTP_CLOCK_ID_LEN.

Param time_base A pointer to the value of timeBaseIndicator of the current grandmaster.

Param last_gm_ph_change A pointer to the value of lastGmPhaseChange received from grandmaster.

Param last_gm_freq_change A pointer to the value of lastGmFreqChange received from the grandmaster.

typedef void (*gptp_port_cb_t)(int port, struct *net_if* *iface, void *user_data)

Callback used while iterating over gPTP ports.

Param port Port number

Param iface Pointer to network interface

Param user_data A valid pointer to user data or NULL

Functions

void gptp_register_phase_dis_cb(struct *gptp_phase_dis_cb* *phase_dis, *gptp_phase_dis_callback_t* cb)

Register a phase discontinuity callback.

Parameters

- phase_dis – Caller specified handler for the callback.
- cb – Callback to register.

void gptp_unregister_phase_dis_cb(struct *gptp_phase_dis_cb* *phase_dis)

Unregister a phase discontinuity callback.

Parameters

- phase_dis – Caller specified handler for the callback.

void gptp_call_phase_dis_cb(void)

Call a phase discontinuity callback function.

int gptp_event_capture(struct *net_ptp_time* *slave_time, bool *gm_present)

Get gPTP time.

Parameters

- slave_time – A pointer to structure where timestamp will be saved.
- gm_present – A pointer to a boolean where status of the presence of a grand master will be saved.

Returns Error code. 0 if no error.

char *gptp_sprint_clock_id(const uint8_t *clk_id, char *output, size_t output_len)

Utility function to print clock id to a user supplied buffer.

Parameters

- clk_id – Clock id
- output – Output buffer
- output_len – Output buffer len

Returns Pointer to output buffer

void gptp_foreach_port(*gptp_port_cb_t* cb, void *user_data)

Go through all the gPTP ports and call callback for each of them.

Parameters

- cb – User-supplied callback function to call
- user_data – User specified data

struct gptp_domain *gptp_get_domain(void)

Get gPTP domain.

This contains all the configuration / status of the gPTP domain.

Returns Pointer to domain or NULL if not found.

void gptp_clk_src_time_invoke(struct *gptp_clk_src_time_invoke_params* *arg)

This interface is used by the ClockSource entity to provide time to the ClockMaster entity of a time-aware system.

Parameters

- arg – Current state and parameters of the ClockSource entity.

struct *gptp_hdr* *gptp_get_hdr(struct *net_pkt* *pkt)

Return pointer to gPTP packet header in network packet.

Parameters

- pkt – Network packet (received or sent)

Returns Pointer to gPTP header.

struct gptp_scaled_ns

#include <gptp.h> Scaled Nanoseconds.

Public Members

int32_t high

High half.

int64_t low

Low half.

struct gptp_uscaled_ns

#include <gptp.h> UScaled Nanoseconds.

Public Members

uint32_t `high`
High half.

uint64_t `low`
Low half.

struct `gptp_port_identity`
#include <gptp.h> Port Identity.

Public Members

uint8_t `clk_id`[GPTP_CLOCK_ID_LEN]
Clock identity of the port.

uint16_t `port_number`
Number of the port.

struct `gptp_flags`
#include <gptp.h>

Public Members

uint8_t `octets`[2]
Byte access.

uint16_t `all`
Whole field access.

struct `gptp_hdr`
#include <gptp.h>

Public Members

uint8_t `message_type`
Type of the message.

uint8_t `transport_specific`
Transport specific, always 1.

uint8_t `ptp_version`
Version of the PTP, always 2.

uint8_t `reserved0`
Reserved field.

uint16_t `message_length`

Total length of the message from the header to the last TLV.

uint8_t `domain_number`

Domain number, always 0.

uint8_t `reserved1`

Reserved field.

struct *gptp_flags* `flags`

Message flags.

int64_t `correction_field`

Correction Field. The content depends of the message type.

uint32_t `reserved2`

Reserved field.

struct *gptp_port_identity* `port_id`

Port Identity of the sender.

uint16_t `sequence_id`

Sequence Id.

uint8_t `control`

Control value. Sync: 0, Follow-up: 2, Others: 5.

int8_t `log_msg_interval`

Message Interval in Log2 for Sync and Announce messages.

struct `gptp_phase_dis_cb`

#include <gptp.h> Phase discontinuity callback structure.

Stores the phase discontinuity callback information. Caller must make sure that the variable pointed by this is valid during the lifetime of registration. Typically this means that the variable cannot be allocated from stack.

Public Members

sys_snode_t `node`

Node information for the slist.

gptp_phase_dis_callback_t `cb`

Phase discontinuity callback.

struct `gptp_clk_src_time_invoke_params`

#include <gptp.h> ClockSourceTime.invoke function parameters.

Parameters passed by ClockSourceTime.invoke function.

Public Members

double `last_gm_freq_change`

Frequency change on the last Time Base Indicator Change.

struct *net_ptp_extended_time* `src_time`

The time this function is invoked.

struct *gptp_scaled_ns* `last_gm_phase_change`

Phase change on the last Time Base Indicator Change.

uint16_t `time_base_indicator`

Time Base - changed only if Phase or Frequency changes.

Precision Time Protocol (PTP) time format

Overview The PTP time struct can store time information in high precision format (nanoseconds). The extended timestamp format can store the time in fractional nanoseconds accuracy. The PTP time format is used in *generic Precision Time Protocol (gPTP)* implementation.

API Reference

group `ptp_time`

Precision Time Protocol time specification.

struct `net_ptp_time`

#include <ptp_time.h> Precision Time Protocol Timestamp format.

This structure represents a timestamp according to the Precision Time Protocol standard.

Seconds are encoded as a 48 bits unsigned integer. Nanoseconds are encoded as a 32 bits unsigned integer.

Public Members

union *net_ptp_time*.[anonymous] `[anonymous]`

Seconds encoded on 48 bits.

uint32_t `nanosecond`

Nanoseconds.

struct `net_ptp_extended_time`

#include <ptp_time.h> Precision Time Protocol Extended Timestamp format.

This structure represents an extended timestamp according to the Precision Time Protocol standard.

Seconds are encoded as 48 bits unsigned integer. Fractional nanoseconds are encoded as 48 bits, their unit is 2*(-16) ns.

Public Members

union *net_ptp_extended_time*.[anonymous] [anonymous]

Seconds encoded on 48 bits.

union *net_ptp_extended_time*.[anonymous] [anonymous]

Fractional nanoseconds on 48 bits.

7.20.7 Controller Area Network

Controller Area Network (CAN)

Overview Controller Area Network is a two-wire serial bus specified by the Bosch CAN Specification, Bosch CAN with Flexible Data-Rate specification and the ISO 11898-1:2003 standard. CAN is mostly known for its application in the automotive domain. However, it is also used in home and industrial automation and other products.

A CAN transceiver is an external device that converts the logic level signals from the CAN controller to the bus-levels. The bus lines are called CAN High (CAN H) and CAN Low (CAN L). The transmit wire from the controller to the transceiver is called CAN TX, and the receive wire is called CAN RX. These wires use the logic levels whereas the bus-level is interpreted differentially between CAN H and CAN L. The bus can be either in the recessive (logical one) or dominant (logical zero) state. The recessive state is when both lines, CAN H and CAN L, at roughly at the same voltage level. This state is also the idle state. To write a dominant bit to the bus, open-drain transistors tie CAN H to Vdd and CAN L to ground. The first and last node use a 120-ohm resistor between CAN H and CAN L to terminate the bus. The dominant state always overrides the recessive state. This structure is called a wired-AND.

Warning: CAN controllers can only initialize when the bus is in the idle (recessive) state for at least 11 recessive bits. Therefore you have to make sure that CAN RX is high, at least for a short time. This is also necessary for loopback mode.

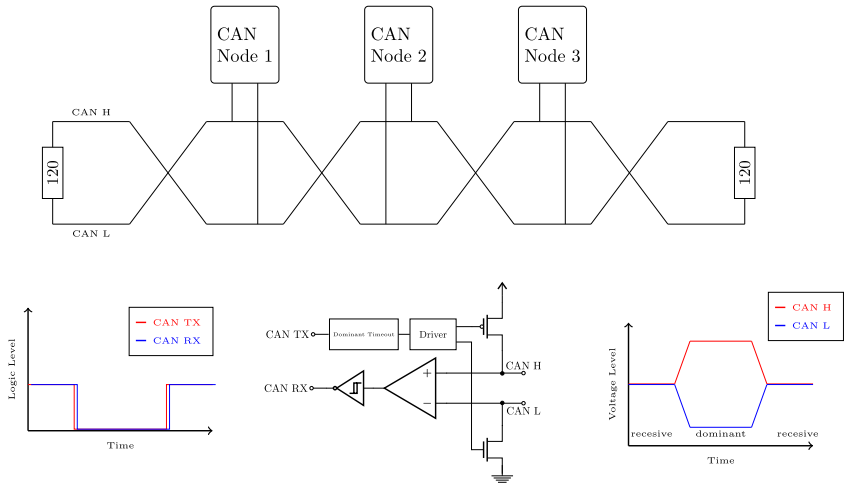


Figure 1: CAN Transceiver, from Logic Levels to Bus Levels. ©Alexander Wachter

The bit-timing as defined in ISO 11898-1:2003 looks as following:

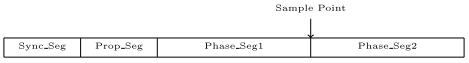


Figure 2: CAN Timing. ©Alexander Wachter

A single bit is split into four segments.

- Sync_Seg: The nodes synchronize at the edge of the Sync_Seg. It is always one time quantum in length.
- Prop_Seg: The signal propagation delay of the bus and other delays of the transceiver and node.
- Phase_Seg1 and Phase_Seg2 :Define the sampling point. The bit is sampled at the end of Phase_Seg1.

The bit-rate is calculated from the time of a time quantum and the values defined above. A bit has the length of Sync_Seg plus Prop_Seg plus Phase_Seg1 plus Phase_Seg2 multiplied by the time of single time quantum. The bit-rate is the inverse of the length of a single bit.

A bit is sampled at the sampling point. The sample point is between Phase_Seg1 and PhaseSeg2 and therefore is a parameter that the user needs to choose. The CiA recommends setting the sample point to 87.5% of the bit.

The resynchronization jump width (SJW) defines the amount of time quantum the sample point can be moved. The sample point is moved when resynchronization is needed.

The timing parameters (SJW, bitrate and sampling point, or bitrate, Prop_Seg, Phase_Seg1and Phase_Seg2) are initially set from the device-tree and can be changed at run-time from the timing-API.

CAN uses so-called identifiers to identify the frame instead of addresses to identify a node. This identifier can either have 11-bit width (Standard or Basic Frame) or 29-bit in case of an Extended Frame. The Zephyr CAN API supports both Standard and Extended identifiers concurrently. A CAN frame starts with a dominant Start Of Frame bit. After that, the identifiers follow. This phase is called the arbitration phase. During the arbitration phase, write collisions are allowed. They resolve by the fact that dominant bits override recessive bits. Nodes monitor the bus and notice when their transmission is being overridden and in case, abort their transmission. This effectively gives lower number identifiers priority over higher number identifiers.

Filters are used to whitelist identifiers that are of interest for the specific node. An identifier that doesn't match any filter is ignored. Filters can either match exactly or a specified part of the identifier. This method is called masking. As an example, a mask with 11 bits set for standard or 29 bits set for extended identifiers must match perfectly. Bits that are set to zero in the mask are ignored when matching an identifier. Most CAN controllers implement a limited number of filters in hardware. The number of filters is also limited in Kconfig to save memory.

Errors may occur during transmission. In case a node detects an erroneous frame, it partially overrides the current frame with an error-frame. Error-frames can either be error passive or error active, depending on the state of the controller. In case the controller is in error active state, it sends six consecutive dominant bits, which is a violation of the stuffing rule that all nodes can detect. The sender may resend the frame right after.

An initialized node can be in one of the following states:

- Error-active
- Error-passive
- Bus-off

After initialization, the node is in the error-active state. In this state, the node is allowed to send active error frames, ACK, and overload frames. Every node has a receive- and transmit-error counter. If either the receive- or the transmit-error counter exceeds 127, the node changes to error-passive state. In this state, the node is not allowed to send error-active frames anymore. If the transmit-error counter increases further to 255, the node changes to the bus-off state. In this state, the node is not allowed to send any dominant bits to the bus. Nodes in the bus-off state may recover after receiving 128 occurrences of 11 concurrent recessive bits.

You can read more about CAN bus in this CAN Wikipedia article.

Zephyr supports following CAN features:

- Standard and Extended Identifers
- Filters with Masking
- Loopback and Silent mode
- Remote Request

Sending The following code snippets show how to send data.

This basic sample sends a CAN frame with standard identifier 0x123 and eight bytes of data. When passing NULL as the callback, as shown in this example, the send function blocks until the frame is sent and acknowledged by at least one other node or an error occurred. The timeout only takes effect on acquiring a mailbox. When a transmitting mailbox is assigned, sending cannot be canceled.

```
struct zcan_frame frame = {
        .id_type = CAN_STANDARD_IDENTIFIER,
        .rtr = CAN_DATAFRAME,
        .id = 0x123,
        .dlc = 8,
        .data = {1,2,3,4,5,6,7,8}
};
const struct device *can_dev;
int ret;

can_dev = device_get_binding("CAN_0");

ret = can_send(can_dev, &frame, K_MSEC(100), NULL, NULL);
if (ret != CAN_TX_OK) {
        LOG_ERR("Sending failed [%d]", ret);
}
```

This example shows how to send a frame with extended identifier 0x1234567 and two bytes of data. The provided callback is called when the message is sent, or an error occurred. Passing *K_FOREVER* to the timeout causes the function to block until a transfer mailbox is assigned to the frame or an error occurred. It does not block until the message is sent like the example above.

```
void tx_irq_callback(int error, void *arg)
{
        char *sender = (char *)arg;

        if (error != 0) {
                LOG_ERR("Sendig failed [%d]\nSender: %s\n", error, sender);
        }
}

int send_function(const struct device *can_dev)
{
        struct zcan_frame frame = {
                .id_type = CAN_EXTENDED_IDENTIFIER,
                .rtr = CAN_DATAFRAME,
                .id = 0x1234567,
                .dlc = 2
        };

        frame.data[0] = 1;
        frame.data[1] = 2;

        return can_send(can_dev, &frame, K_FOREVER, tx_irq_callback, "Sender 1");
}
```

Receiving Frames are only received when they match a filter. The following code snippets show how to receive frames by attaching filters.

Here we have an example for a receiving callback. It is used for *can_attach_isr* or *can_attach_workq*. The argument arg is passed when the filter is attached.

```
void rx_callback_function(struct zcan_frame *frame, void *arg)
{
        ... do something with the frame ...
}
```

The following snippet shows how to attach a filter with an interrupt callback. It is the most efficient but also the most critical way to receive messages. The callback function is called from an interrupt context, which means that the callback function should be as short as possible and must not block. Attaching ISRs is not allowed from userspace context.

The filter for this example is configured to match the identifier 0x123 exactly.

```
const struct zcan_filter my_filter = {
        .id_type = CAN_STANDARD_IDENTIFIER,
        .rtr = CAN_DATAFRAME,
        .id = 0x123,
        .rtr_mask = 1,
        .id_mask = CAN_STD_ID_MASK
};
int filter_id;
const struct device *can_dev;

can_dev = device_get_binding("CAN_0");

filter_id = can_attach_isr(can_dev, rx_callback_function, callback_arg, &my_filter);
if (filter_id < 0) {
```

(continues on next page)

(continued from previous page)

```
  LOG_ERR("Unable to attach isr [%d]", filter_id);
}
```

This example shows how to attach a callback from a work-queue. In contrast to the *can_attach_isr* function, here the callback is called from the work-queue provided. In this case, it is the system work queue. Blocking is generally allowed in the callback but could result in a frame backlog when it is not limited. For the reason of a backlog, a ring-buffer is applied for every attached filter. The size of this buffer can be adjusted in Kconfig. This function is not yet callable from userspace context but will be in the future.

The filter for this example is configured to match a filter range from 0x120 to x12f.

```
const struct zcan_filter my_filter = {
        .id_type = CAN_STANDARD_IDENTIFIER,
        .rtr = CAN_DATAFRAME,
        .id = 0x120,
        .rtr_mask = 1,
        .id_mask = 0x7F0
};
struct zcan_work rx_work;
int filter_id;
const struct device *can_dev;

can_dev = device_get_binding("CAN_0");

filter_id = can_attach_workq(can_dev, &k_sys_work_q, &rx_work, callback_arg, callback_
↪arg, &my_filter);
if (filter_id < 0) {
  LOG_ERR("Unable to attach isr [%d]", filter_id);
}
```

Here an example for *can_attach_msgq* is shown. With this function, it is possible to receive frames synchronously. This function can be called from userspace context. The size of the message queue should be as big as the expected backlog.

The filter for this example is configured to match the extended identifier 0x1234567 exactly.

```
const struct zcan_filter my_filter = {
        .id_type = CAN_EXTENDED_IDENTIFIER,
        .rtr = CAN_DATAFRAME,
        .id = 0x1234567,
        .rtr_mask = 1,
        .id_mask = CAN_EXT_ID_MASK
};
CAN_DEFINE_MSGQ(my_can_msgq, 2);
struct zcan_frame rx_frame;
int filter_id;
const struct device *can_dev;

can_dev = device_get_binding("CAN_0");

filter_id = can_attach_msgq(can_dev, &my_can_msgq, &my_filter);
if (filter_id < 0) {
  LOG_ERR("Unable to attach isr [%d]", filter_id);
  return;
}

while (true) {
```

(continues on next page)

(continued from previous page)

```
  k_msgq_get(&my_can_msgq, &rx_frame, K_FOREVER);
  ... do something with the frame ...
}
```

can_detach removes the given filter.

```
can_detach(can_dev, filter_id);
```

Setting the bitrate The bitrate and sampling point is initially set at runtime. To change it from the application, one can use the *can_set_timing* API. This function takes three arguments. The first timing parameter sets the timing for classic CAN and arbitration phase for CAN-FD. The second parameter sets the timing of the data phase for CAN-FD. For classic CAN, you can use only the first parameter and put NULL to the second one. The *can_calc_timing* function can calculate timing from a bitrate and sampling point in permille. The following example sets the bitrate to 250k baud with the sampling point at 87.5%.

```
struct can_timing timing;
const struct device *can_dev;
int ret;

can_dev = device_get_binding("CAN_0");

ret = can_calc_timing(can_dev, &timing, 250000, 875);
if (ret > 0) {
  LOG_INF("Sample-Point error: %d", ret);
}

if (ret < 0) {
  LOG_ERR("Failed to calc a valid timing");
  return;
}

ret = can_set_timing(can_dev, &timing, NULL);
if (ret != 0) {
  LOG_ERR("Failed to set timing");
}
```

SocketCAN Zephyr additionally supports SocketCAN, a BSD socket implementation of the Zephyr CAN API. SocketCAN brings the convenience of the well-known BSD Socket API to Controller Area Networks. It is compatible with the Linux SocketCAN implementation, where many other high-level CAN projects build on top. Note that frames are routed to the network stack instead of passed directly, which adds some computation and memory overhead.

Samples We have two ready-to-build samples demonstrating use of the Zephyr CAN API Zephyr CAN sample and SocketCAN sample.

API Reference

group `can_interface`

CAN Interface.

Defines

CAN_EX_ID

CAN_MAX_STD_ID

CAN_STD_ID_MASK

CAN_EXT_ID_MASK

CAN_MAX_DLC

CANFD_MAX_DLC

CAN_MAX_DLEN

CAN_TX_OK
: send successfully

CAN_TX_ERR
: general send error

CAN_TX_ARB_LOST
: bus arbitration lost during sending

CAN_TX_BUS_OFF
: controller is in bus off state

CAN_TX_UNKNOWN
: unexpected error

CAN_TX_EINVAL
: invalid parameter

CAN_NO_FREE_FILTER
: attach_* failed because there is no unused filter left

CAN_TIMEOUT
: operation timed out

CAN_DEFINE_MSGQ(name, size)
: Statically define and initialize a can message queue.

  The message queue's ring buffer contains space for *size* messages.

  Parameters

  - name – Name of the message queue.
  - size – Number of can messages.

CAN_SJW_NO_CHANGE
: SWJ value to indicate that the SJW should not be changed

CONFIG_CAN_WORKQ_FRAMES_BUF_CNT

Typedefs

typedef uint32_t canid_t

typedef void (*can_tx_callback_t)(int error, void *arg)

Define the application callback handler function signature.

Param error status of the performed send operation

Param arg argument that was passed when the message was sent

typedef void (*can_rx_callback_t)(struct *zcan_frame* *msg, void *arg)

Define the application callback handler function signature for receiving.

Param msg received message

Param arg argument that was passed when the filter was attached

typedef void (*can_state_change_isr_t)(enum *can_state* state, struct *can_bus_err_cnt* err_cnt)

Defines the state change isr handler function signature.

Param state state of the node

Param err_cnt struct with the error counter values

typedef int (*can_set_timing_t)(const struct *device* *dev, const struct *can_timing* *timing, const struct *can_timing* *timing_data)

typedef int (*can_set_mode_t)(const struct *device* *dev, enum *can_mode* mode)

typedef int (*can_send_t)(const struct *device* *dev, const struct *zcan_frame* *msg, *k_timeout_t* timeout, *can_tx_callback_t* callback_isr, void *callback_arg)

typedef int (*can_attach_msgq_t)(const struct *device* *dev, struct *k_msgq* *msg_q, const struct *zcan_filter* *filter)

typedef int (*can_attach_isr_t)(const struct *device* *dev, *can_rx_callback_t* isr, void *callback_arg, const struct *zcan_filter* *filter)

typedef void (*can_detach_t)(const struct *device* *dev, int filter_id)

typedef int (*can_recover_t)(const struct *device* *dev, *k_timeout_t* timeout)

typedef enum *can_state* (*can_get_state_t)(const struct *device* *dev, struct *can_bus_err_cnt* *err_cnt)

typedef void (*can_register_state_change_isr_t)(const struct *device* *dev, *can_state_change_isr_t* isr)

typedef int (*can_get_core_clock_t)(const struct *device* *dev, uint32_t *rate)

Enums

enum `can_ide`

can_ide enum Define if the message has a standard (11bit) or extended (29bit) identifier

Values:

enumerator `CAN_STANDARD_IDENTIFIER`

enumerator `CAN_EXTENDED_IDENTIFIER`

enum `can_rtr`

can_rtr enum Define if the message is a data or remote frame

Values:

enumerator `CAN_DATAFRAME`

enumerator `CAN_REMOTEREQUEST`

enum `can_mode`

can_mode enum Defines the mode of the can controller

Values:

enumerator `CAN_NORMAL_MODE`

enumerator `CAN_SILENT_MODE`

enumerator `CAN_LOOPBACK_MODE`

enumerator `CAN_SILENT_LOOPBACK_MODE`

enum `can_state`

can_state enum Defines the possible states of the CAN bus

Values:

enumerator `CAN_ERROR_ACTIVE`

enumerator `CAN_ERROR_PASSIVE`

enumerator `CAN_BUS_OFF`

enumerator `CAN_BUS_UNKNOWN`

Functions

static inline uint8_t `can_dlc_to_bytes`(uint8_t dlc)

Convert the DLC to the number of bytes.

This function converts a the Data Length Code to the number of bytes.

Parameters

- `dlc` – The Data Length Code

Return values `Number` – of bytes

static inline uint8_t `can_bytes_to_dlc`(uint8_t num_bytes)

Convert a number of bytes to the DLC.

This function converts a number of bytes to the Data Length Code

Parameters

- `num_bytes` – The number of bytes

Return values `The` – DLC

int `can_send`(const struct *device* *dev, const struct *zcan_frame* *msg, *k_timeout_t* timeout, *can_tx_callback_t* callback_isr, void *callback_arg)

Perform data transfer to CAN bus.

This routine provides a generic interface to perform data transfer to the can bus. Use *can_write()* for simple write.

-

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `msg` – Message to transfer.
- `timeout` – Waiting for empty tx mailbox timeout or K_FOREVER.
- `callback_isr` – Is called when message was sent or a transmission error occurred. If NULL, this function is blocking until message is sent. This must be NULL if called from user mode.
- `callback_arg` – This will be passed whenever the isr is called.

Return values

- `0` – If successful.
- `CAN_TX_*` – on failure.

static inline int `can_write`(const struct *device* *dev, const uint8_t *data, uint8_t length, uint32_t id, enum *can_rtr* rtr, *k_timeout_t* timeout)

Write a set amount of data to the can bus.

This routine writes a set amount of data synchronously.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `data` – Data to send.
- `length` – Number of bytes to write (max. 8).
- `id` – Identifier of the can message.
- `rtr` – Send remote transmission request or data frame
- `timeout` – Waiting for empty tx mailbox timeout or K_FOREVER

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-EINVAL` – if length > 8.

int `can_attach_workq`(const struct *device* *dev, struct *k_work_q* *work_q, struct *zcan_work* *work, *can_rx_callback_t* callback, void *callback_arg, const struct *zcan_filter* *filter)

Attach a CAN work queue to a single or group of identifiers.

This routine attaches a work queue to identifiers specified by a filter. Whenever the filter matches, the message is pushed to the buffer of the *zcan_work* structure and the work element is put to the workqueue. If a message passes more than one filter the priority of the match is hardware dependent. A CAN work queue can be attached to more than one filter. The work queue must be initialized before and the caller must have appropriate permissions on it.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `work_q` – Pointer to the already initialized work queue.
- `work` – Pointer to a *zcan_work*. The work will be initialized.
- `callback` – This function is called by workq whenever a message arrives.
- `callback_arg` – Is passed to the callback when called.
- `filter` – Pointer to a *zcan_filter* structure defining the id filtering.

Return values

- `filter_id` – on success.
- `CAN_NO_FREE_FILTER` – if there is no filter left.

int `can_attach_msgq`(const struct *device* *dev, struct *k_msgq* *msg_q, const struct *zcan_filter* *filter)

Attach a message queue to a single or group of identifiers.

This routine attaches a message queue to identifiers specified by a filter. Whenever the filter matches, the message is pushed to the queue If a message passes more than one filter the priority of the match is hardware dependent. A message queue can be attached to more than one filter. The message queue must me initialized before, and the caller must have appropriate permissions on it.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `msg_q` – Pointer to the already initialized message queue.
- `filter` – Pointer to a *zcan_filter* structure defining the id filtering.

Return values

- `filter_id` – on success.
- `CAN_NO_FREE_FILTER` – if there is no filter left.

static inline int `can_attach_isr`(const struct *device* *dev, *can_rx_callback_t* isr, void *callback_arg, const struct *zcan_filter* *filter)

Attach an isr callback function to a single or group of identifiers.

This routine attaches an isr callback to identifiers specified by a filter. Whenever the filter matches, the callback function is called with isr context. If a message passes more than one

filter the priority of the match is hardware dependent. A callback function can be attached to more than one filter.

-

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `isr` – Callback function pointer.
- `callback_arg` – This will be passed whenever the isr is called.
- `filter` – Pointer to a *zcan_filter* structure defining the id filtering.

Return values

- `filter_id` – on success.
- `CAN_NO_FREE_FILTER` – if there is no filter left.

void `can_detach`(const struct *device* *dev, int filter_id)

Detach an isr or message queue from the identifier filtering.

This routine detaches an isr callback or message queue from the identifier filtering.

-

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `filter_id` – filter id returned by can_attach_isr or can_attach_msgq.

Return values `none` –

int `can_get_core_clock`(const struct *device* *dev, uint32_t *rate)

Read the core clock value.

Returns the core clock value. One time quantum is 1/core clock.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `rate` – **[out]** controller clock rate

Return values

- `0` – on success
- `negative` – on error

int `can_calc_timing`(const struct *device* *dev, struct *can_timing* *res, uint32_t bitrate, uint16_t sample_pnt)

Calculate timing parameters from bitrate and sample point.

Calculate the timing parameters from a given bitrate in bits/s and the sampling point in permill (1/1000) of the entire bit time. The bitrate must alway match perfectly. If no result can be given for the, give parameters, -EINVAL is returned. The sample_pnt does not always match perfectly. The algorithm tries to find the best match possible.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `res` – Result is written into the *can_timing* struct provided.
- `bitrate` – Target bitrate in bits/s
- `sample_pnt` – Sampling point in permill of the entire bit time.

Return values

- `Positive` – sample point error on success
- `-EINVAL` – if there is no solution for the desired values
- `-EIO` – if core_clock is not available

int `can_calc_prescaler`(const struct *device* *dev, struct *can_timing* *timing, uint32_t bitrate)

Fill in the prescaler value for a given bitrate and timing.

Fill the prescaler value in the timing struct. sjw, prop_seg, phase_seg1 and phase_seg2 must be given. The returned bitrate error is reminder of the devision of the clockrate by the bitrate times the timing segments.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `timing` – Result is written into the *can_timing* struct provided.
- `bitrate` – Target bitrate.

Return values

- `bitrate` – error
- `negative` – on error

int `can_set_mode`(const struct *device* *dev, enum *can_mode* mode)

Set the controller to the given mode.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mode` – Operation mode

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.

int `can_set_timing`(const struct *device* *dev, const struct *can_timing* *timing, const struct *can_timing* *timing_data)

Configure timing of a host controller.

If the sjw equals CAN_SJW_NO_CHANGE, the sjw parameter is not changed.

The second parameter timing_data is only relevant for CAN-FD. If the controller does not support CAN-FD or the FD mode is not enabled, this parameter is ignored.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `timing` – Bus timings
- `timing_data` – Bus timings for data phase (CAN-FD only)

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.

static inline int `can_set_bitrate`(const struct *device* *dev, uint32_t bitrate, uint32_t bitrate_data)

Set the bitrate of the CAN controller.

The second parameter bitrate_data is only relevant for CAN-FD. If the controller does not support CAN-FD or the FD mode is not enabled, this parameter is ignored. The sample point is set to the CiA DS 301 reccommended value of 87.5%

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `bitrate` – Desired arbitration phase bitrate
- `bitrate_data` – Desired data phase bitrate

Return values

- `0` – If successful.
- `-EINVAL` – bitrate cannot be reached.
- `-EIO` – General input / output error, failed to set bitrate.

static inline int `can_configure`(const struct *device* *dev, enum *can_mode* mode, uint32_t bitrate)

Configure operation of a host controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `mode` – Operation mode
- `bitrate` – bus-speed in Baud/s

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.

enum *can_state* `can_get_state`(const struct *device* *dev, struct *can_bus_err_cnt* *err_cnt)

Get current state.

Returns the actual state of the CAN controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `err_cnt` – Pointer to the err_cnt destination structure or NULL.

Return values `state` –

int `can_recover`(const struct *device* *dev, *k_timeout_t* timeout)

Recover from bus-off state.

Recover the CAN controller from bus-off state to error-active state.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `timeout` – Timeout for waiting for the recovery or K_FOREVER.

Return values

- `0` – on success.
- `CAN_TIMEOUT` – on timeout.

static inline void `can_register_state_change_isr`(const struct *device* *dev, *can_state_change_isr_t* isr)

Register an ISR callback for state change interrupt.

Only one callback can be registered per controller. Calling this function again, overrides the previous call.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

- `isr` – Pointer to ISR

static inline void `can_copy_frame_to_zframe`(const struct *can_frame* *frame, struct *zcan_frame* *zframe)

Converter that translates between *can_frame* and *zcan_frame* structs.

Parameters

- `frame` – Pointer to *can_frame* struct.
- `zframe` – Pointer to *zcan_frame* struct.

static inline void `can_copy_zframe_to_frame`(const struct *zcan_frame* *zframe, struct *can_frame* *frame)

Converter that translates between *zcan_frame* and *can_frame* structs.

Parameters

- `zframe` – Pointer to *zcan_frame* struct.
- `frame` – Pointer to *can_frame* struct.

static inline void `can_copy_filter_to_zfilter`(const struct *can_filter* *filter, struct *zcan_filter* *zfilter)

Converter that translates between *can_filter* and zcan_frame_filter structs.

Parameters

- `filter` – Pointer to *can_filter* struct.
- `zfilter` – Pointer to zcan_frame_filter struct.

static inline void `can_copy_zfilter_to_filter`(const struct *zcan_filter* *zfilter, struct *can_filter* *filter)

Converter that translates between *zcan_filter* and *can_filter* structs.

Parameters

- `zfilter` – Pointer to *zcan_filter* struct.
- `filter` – Pointer to *can_filter* struct.

struct `can_frame`

#include <can.h> CAN frame structure that is compatible with Linux. This is mainly used by Socket CAN code.

Used to pass CAN messages from userspace to the socket CAN and vice versa.

Public Members

canid_t `can_id`

32 bit CAN_ID + EFF/RTR/ERR flags

uint8_t `can_dlc`

The length of the message

uint8_t `data`[8]

The message data

struct `can_filter`

#include <can.h> CAN filter that is compatible with Linux. This is mainly used by Socket CAN code.

A filter matches, when "received_can_id & mask == can_id & mask"

struct `zcan_frame`

#include <can.h> CAN message structure.

Used to pass can messages from userspace to the driver and from driver to userspace

Public Members

uint32_t `id`

Message identifier

uint32_t `fd`

Frame is in the CAN-FD frame format

uint32_t `rtr`

Set the message to a transmission request instead of data frame use can_rtr enum for assignment

uint32_t `id_type`

Indicates the identifier type (standard or extended) use can_ide enum for assignment

uint8_t `dlc`

The length of the message (max. 8) in byte

uint8_t `brs`

Baud Rate Switch. Frame transfer with different timing during the data phase. Only valid for CAN-FD

uint8_t `res`

Reserved for future flags

union *zcan_frame*.[anonymous] `[anonymous]`

The frame payload data.

struct `zcan_filter`

#include <can.h> CAN filter structure.

Used to pass can identifier filter information to the driver. rtr_mask and *_id_mask are used to mask bits of the rtr and id fields. If the mask bit is 0, the value of the corresponding bit in the id or rtr field don't care for the filter matching.

Public Members

uint32_t `id`

target state of the identifier

uint32_t `rtr`

target state of the rtr bit

uint32_t `id_type`

Indicates the identifier type (standard or extended) use can_ide enum for assignment

uint32_t `id_mask`

identifier mask

uint32_t `rtr_mask`

rtr bit mask

struct `can_bus_err_cnt`

#include <can.h> can bus error count structure

Used to pass the bus error counters to userspace

struct `can_timing`

#include <can.h> canbus timings

Used to pass bus timing values to the config and bitrate calculator function.

The propagation segment represents the time of the signal propagation. Phase segment 1 and phase segment 2 define the sampling point. prop_seg and phase_seg1 affect the sampling-point in the same way and some controllers only have a register for the sum of those two. The sync segment always has a length of 1 tq +———+——-—+————+————+ |sync_seg | prop_seg | phase_seg1 | phase_seg2 | +———+——-—+————+————+ ^ Sampling-Point 1 tq (time quantum) has the length of 1/(core_clock / prescaler) The bitrate is defined by the core clock divided by the prescaler and the sum of the segments. br = (core_clock / prescaler) / (1 + prop_seg + phase_seg1 + phase_seg2) The resynchronization jump width (SJW) defines the amount of time quantum the sample point can be moved. The sample point is moved when resynchronization is needed.

Public Members

uint16_t `sjw`

Synchronisation jump width

uint16_t `prop_seg`

Propagation Segment

uint16_t `phase_seg1`

Phase Segment 1

uint16_t `phase_seg2`

Phase Segment 2

uint16_t `prescaler`

Prescaler value

struct `can_frame_buffer`

#include <can.h>

struct `zcan_work`

#include <can.h> CAN work structure.

Used to attach a work queue to a filter.

struct `can_driver_api`

#include <can.h>

ISO-TP Transport Protocol

Overview ISO-TP is a transport protocol defined in the ISO-Standard ISO15765-2 Road vehicles - Diagnostic communication over Controller Area Network (DoCAN). Part2: Transport protocol and network layer services. As its name already implies, it is originally designed, and still used in road vehicle diagnostic over Controller Area Networks. Nevertheless, it's not limited to applications in road vehicles or the automotive domain.

This transport protocol extends the limited payload data size for classical CAN (8 bytes) and CAN-FD (64 bytes) to theoretically four gigabytes. Additionally, it adds a flow control mechanism to influence the sender's behavior. ISO-TP segments packets into small fragments depending on the payload size of the CAN frame. The header of those segments is called Protocol Control Information (PCI).

Packets smaller or equal to seven bytes on Classical CAN are called single-frames (SF). They don't need to fragment and do not have any flow-control.

Packets larger than that are segmented into a first-frame (FF) and as many consecutive-frames as required. The FF contains information about the length of the entire payload data and additionally, the first few bytes of payload data. The receiving peer sends back a flow-control-frame (FC) to either deny, postpone, or accept the following consecutive frames. The FC also defines the conditions of sending, namely the block-size (BS) and the minimum separation time between frames (STmin). The block size defines how many CF the sender is allowed to send, before he has to wait for another FC.

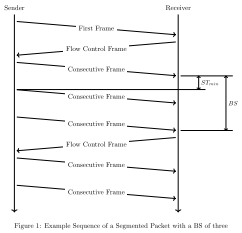


Figure 1: Example Sequence of a Segmented Packet with a BS of three

API Reference

group `can_isotp`

CAN ISO-TP Interf.

Defines

ISOTP_N_OK
: Completed successfully

ISOTP_N_TIMEOUT_A
: Ar/As has timed out

ISOTP_N_TIMEOUT_BS
: Reception of next FC has timed out

ISOTP_N_TIMEOUT_CR
: Cr has timed out

ISOTP_N_WRONG_SN
: Unexpected sequence number

ISOTP_N_INVALID_FS
: Invalid flow status received

ISOTP_N_UNEXP_PDU
: Unexpected PDU received

ISOTP_N_WFT_OVRN
: Maximum number of WAIT flowStatus PDUs exceeded

ISOTP_N_BUFFER_OVERFLW
: FlowStatus OVFLW PDU was received

ISOTP_N_ERROR
: General error

ISOTP_NO_FREE_FILTER
: Implementation specific errors Can't bind or send because the CAN device has no filter left

ISOTP_NO_NET_BUF_LEFT
: No net buffer left to allocate

ISOTP_NO_BUF_DATA_LEFT
: Not sufficient space in the buffer left for the data

ISOTP_NO_CTX_LEFT
: No context buffer left to allocate

ISOTP_RECV_TIMEOUT
: Timeout for recv

ISOTP_FIXED_ADDR_SA_POS
: Position of fixed source address (SA)

ISOTP_FIXED_ADDR_SA_MASK

Mask to obtain fixed source address (SA)

ISOTP_FIXED_ADDR_TA_POS

Position of fixed target address (TA)

ISOTP_FIXED_ADDR_TA_MASK

Mask to obtain fixed target address (TA)

ISOTP_FIXED_ADDR_PRIO_POS

Position of priority in fixed addressing mode

ISOTP_FIXED_ADDR_PRIO_MASK

Mask for priority in fixed addressing mode

ISOTP_FIXED_ADDR_RX_MASK

Typedefs

typedef void (*isotp_tx_callback_t)(int error_nr, void *arg)

Functions

int isotp_bind(struct isotp_recv_ctx *ctx, const struct *device* *can_dev, const struct *isotp_msg_id* *rx_addr, const struct *isotp_msg_id* *tx_addr, const struct *isotp_fc_opts* *opts, *k_timeout_t* timeout)

Bind an address to a receiving context.

This function binds an RX and TX address combination to an RX context. When data arrives from the specified address, it is buffered and can be read by calling isotp_recv. When calling this routine, a filter is applied in the CAN device, and the context is initialized. The context must be valid until calling unbind.

Parameters

- ctx – Context to store the internal states.
- can_dev – The CAN device to be used for sending and receiving.
- rx_addr – Identifier for incoming data.
- tx_addr – Identifier for FC frames.
- opts – Flow control options.
- timeout – Timeout for FF SF buffer allocation.

Return values

- ISOTP_N_OK – on success
- ISOTP_NO_FREE_FILTER – if CAN device has no filters left.

void `isotp_unbind`(struct isotp_recv_ctx *ctx)

Unbind a context from the interface.

This function removes the binding from isotp_bind. The filter is detached from the CAN device, and if a transmission is ongoing, buffers are freed. The context can be discarded safely after calling this function.

Parameters

- `ctx` – Context that should be unbound.

int `isotp_recv`(struct isotp_recv_ctx *ctx, uint8_t *data, size_t len, *k_timeout_t* timeout)

Read out received data from fifo.

This function reads the data from the receive FIFO of the context. It blocks if the FIFO is empty. If an error occurs, the function returns a negative number and leaves the data buffer unchanged.

Parameters

- `ctx` – Context that is already bound.
- `data` – Pointer to a buffer where the data is copied to.
- `len` – Size of the buffer.
- `timeout` – Timeout for incoming data.

Return values

- `Number` – of bytes copied on success
- `ISOTP_WAIT_TIMEOUT` – when "timeout" timed out
- `ISOTP_N_*` – on error

int `isotp_recv_net`(struct isotp_recv_ctx *ctx, struct *net_buf* **buffer, *k_timeout_t* timeout)

Get the net buffer on data reception.

This function reads incoming data into net-buffers. It blocks until the entire packet is received, BS is reached, or an error occurred. If BS was zero, the data is in a single *net_buf*. Otherwise, the data is fragmented in chunks of BS size. The net-buffers are referenced and must be freed with net_buf_unref after the data is processed.

Parameters

- `ctx` – Context that is already bound.
- `buffer` – Pointer where the *net_buf* pointer is written to.
- `timeout` – Timeout for incoming data.

Return values

- `Remaining` – data length for this transfer if BS > 0, 0 for BS = 0
- `ISOTP_WAIT_TIMEOUT` – when "timeout" timed out
- `ISOTP_N_*` – on error

int `isotp_send`(struct isotp_send_ctx *ctx, const struct *device* *can_dev, const uint8_t *data, size_t len, const struct *isotp_msg_id* *tx_addr, const struct *isotp_msg_id* *rx_addr, *isotp_tx_callback_t* complete_cb, void *cb_arg)

Send data.

This function is used to send data to a peer that listens to the tx_addr. An internal work-queue is used to transfer the segmented data. Data and context must be valid until the transmission has finished. If a complete_cb is given, this function is non-blocking, and the callback is called on completion with the return value as a parameter.

Parameters

- `ctx` – Context to store the internal states.
- `can_dev` – The CAN device to be used for sending and receiving.
- `data` – Data to be sent.
- `len` – Length of the data to be sent.
- `rx_addr` – Identifier for FC frames.
- `tx_addr` – Identifier for outgoing frames the receiver listens on.
- `complete_cb` – Function called on completion or NULL.
- `cb_arg` – Argument passed to the complete callback.

Return values

- `ISOTP_N_OK` – on success
- `ISOTP_N_*` – on error

struct `isotp_msg_id`

#include <isotp.h> ISO-TP message id struct.

Used to pass addresses to the bind and send functions.

Public Members

union *isotp_msg_id*.[anonymous] `[anonymous]`

CAN identifier

If ISO-TP fixed addressing is used, isotp_bind ignores SA and priority sections and modifies TA section in flow control frames.

uint8_t `ext_addr`

ISO-TP extended address (if used)

uint8_t `id_type`

Indicates the CAN identifier type (standard or extended)

uint8_t `use_ext_addr`

Indicates if ISO-TP extended addressing is used

uint8_t `use_fixed_addr`

Indicates if ISO-TP fixed addressing (acc. to SAE J1939) is used

struct `isotp_fc_opts`

#include <isotp.h> ISO-TP frame control options struct.

Used to pass the options to the bind and send functions.

Public Members

uint8_t `bs`

Block size. Number of CF PDUs before next CF is sent

uint8_t `stmin`

Minimum separation time. Min time between frames

7.20.8 Generic GSM Modem

Overview

The generic GSM modem driver allows the user to connect Zephyr to a GSM modem which provides a data connection to cellular operator's network. The Zephyr uses *PPP (Point-to-Point Protocol)* to connect to the GSM modem using UART. Note that some cellular modems have proprietary offloading support using AT commands, but usually those modems also support 3GPP standards and provide PPP connection to them. See GSM modem sample application how to setup Zephyr to use the GSM modem.

The GSM muxing, that is defined in GSM 07.10, and which allows mixing of AT commands and PPP traffic, is also supported in this version of Zephyr. One needs to enable `CONFIG_GSM_MUX` and `CONFIG_UART_MUX` configuration options to enable muxing.

7.21 Peripherals

7.21.1 ADC

Overview

API Reference

group `adc_interface`

ADC driver APIs.

Typedefs

typedef enum *adc_action* (*`adc_sequence_callback`)(const struct *device* *dev, const struct *adc_sequence* *sequence, uint16_t sampling_index)

Type definition of the optional callback function to be called after a requested sampling is done.

Param dev Pointer to the device structure for the driver instance.

Param sequence Pointer to the sequence structure that triggered the sampling. This parameter points to a copy of the structure that was supplied to the call that started the sampling sequence, thus it cannot be used with the *CONTAINER_OF()* macro to retrieve some other data associated with the sequence. Instead, the *adc_sequence_options::user_data* field should be used for such purpose.

Param sampling_index Index (0-65535) of the sampling done.

Return Action to be performed by the driver. See *adc_action*.

typedef int (*`adc_api_channel_setup`)(const struct *device* *dev, const struct *adc_channel_cfg* *channel_cfg)

Type definition of ADC API function for configuring a channel. See *adc_channel_setup()* for argument descriptions.

typedef int (*adc_api_read)(const struct *device* *dev, const struct *adc_sequence* *sequence)

Type definition of ADC API function for setting a read request. See *adc_read()* for argument descriptions.

typedef int (*adc_api_read_async)(const struct *device* *dev, const struct *adc_sequence* *sequence, struct *k_poll_signal* *async)

Type definition of ADC API function for setting an asynchronous read request. See *adc_read_async()* for argument descriptions.

Enums

enum adc_gain

ADC channel gain factors.

Values:

enumerator ADC_GAIN_1_6

x 1/6.

enumerator ADC_GAIN_1_5

x 1/5.

enumerator ADC_GAIN_1_4

x 1/4.

enumerator ADC_GAIN_1_3

x 1/3.

enumerator ADC_GAIN_1_2

x 1/2.

enumerator ADC_GAIN_2_3

x 2/3.

enumerator ADC_GAIN_1

x 1.

enumerator ADC_GAIN_2

x 2.

enumerator ADC_GAIN_3

x 3.

enumerator ADC_GAIN_4

x 4.

enumerator ADC_GAIN_6

x 6.

enumerator ADC_GAIN_8
x 8.

enumerator ADC_GAIN_12
x 12.

enumerator ADC_GAIN_16
x 16.

enumerator ADC_GAIN_24
x 24.

enumerator ADC_GAIN_32
x 32.

enumerator ADC_GAIN_64
x 64.

enumerator ADC_GAIN_128
x 128.

enum adc_reference
ADC references.

Values:

enumerator ADC_REF_VDD_1
VDD.

enumerator ADC_REF_VDD_1_2
VDD/2.

enumerator ADC_REF_VDD_1_3
VDD/3.

enumerator ADC_REF_VDD_1_4
VDD/4.

enumerator ADC_REF_INTERNAL
Internal.

enumerator ADC_REF_EXTERNAL0
External, input 0.

enumerator ADC_REF_EXTERNAL1
External, input 1.

enum adc_action
Action to be performed after a sampling is done.

Values:

enumerator ADC_ACTION_CONTINUE = 0

The sequence should be continued normally.

enumerator ADC_ACTION_REPEAT

The sampling should be repeated. New samples or sample should be read from the ADC and written in the same place as the recent ones.

enumerator ADC_ACTION_FINISH

The sequence should be finished immediately.

Functions

int adc_gain_invert(enum *adc_gain* gain, int32_t *value)

Invert the application of gain to a measurement value.

For example, if the gain passed in is ADC_GAIN_1_6 and the referenced value is 10, the value after the function returns is 60.

Parameters

- gain – the gain used to amplify the input signal.
- value – a pointer to a value that initially has the effect of the applied gain but has that effect removed when this function successfully returns. If the gain cannot be reversed the value remains unchanged.

Return values

- 0 – if the gain was successfully reversed
- -EINVAL – if the gain could not be interpreted

static inline int adc_raw_to_millivolts(int32_t ref_mv, enum *adc_gain* gain, uint8_t resolution, int32_t *valp)

Convert a raw ADC value to millivolts.

This function performs the necessary conversion to transform a raw ADC measurement to a voltage in millivolts.

Parameters

- ref_mv – the reference voltage used for the measurement, in millivolts. This may be from *adc_ref_internal()* or a known external reference.
- gain – the ADC gain configuration used to sample the input
- resolution – the number of bits in the absolute value of the sample. For differential sampling this may be one less than the resolution in struct *adc_sequence*.
- valp – pointer to the raw measurement value on input, and the corresponding millivolt value on successful conversion. If conversion fails the stored value is left unchanged.

Return values

- 0 – on successful conversion
- -EINVAL – if the gain is not reversible

int adc_channel_setup(const struct *device* *dev, const struct *adc_channel_cfg* *channel_cfg)

Configure an ADC channel.

It is required to call this function and configure each channel before it is selected for a read request.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel_cfg` – Channel configuration.

Return values

- `0` – On success.
- `-EINVAL` – If a parameter with an invalid value has been provided.

int `adc_read`(const struct *device* *dev, const struct *adc_sequence* *sequence)

Set a read request.

If invoked from user mode, any sequence struct options for callback must be NULL.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `sequence` – Structure specifying requested sequence of samplings.

Return values

- `0` – On success.
- `-EINVAL` – If a parameter with an invalid value has been provided.
- `-ENOMEM` – If the provided buffer is to small to hold the results of all requested samplings.
- `-ENOTSUP` – If the requested mode of operation is not supported.
- `-EBUSY` – If another sampling was triggered while the previous one was still in progress. This may occur only when samplings are done with intervals, and it indicates that the selected interval was too small. All requested samples are written in the buffer, but at least some of them were taken with an extra delay compared to what was scheduled.

int `adc_read_async`(const struct *device* *dev, const struct *adc_sequence* *sequence, struct *k_poll_signal* *async)

Set an asynchronous read request.

If invoked from user mode, any sequence struct options for callback must be NULL.

Note: This function is available only if `CONFIG_ADC_ASYNC` is selected.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `sequence` – Structure specifying requested sequence of samplings.
- `async` – Pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Returns 0 on success, negative error code otherwise. See *adc_read()* for a list of possible error codes.

static inline uint16_t `adc_ref_internal`(const struct *device* *dev)

Get the internal reference voltage.

Returns the voltage corresponding to *ADC_REF_INTERNAL*, measured in millivolts.

Returns a positive value is the reference voltage value. Returns zero if reference voltage information is not available.

struct `adc_channel_cfg`

#include <adc.h> Structure for specifying the configuration of an ADC channel.

Public Members

enum *adc_gain* `gain`

Gain selection.

enum *adc_reference* `reference`

Reference selection.

uint16_t `acquisition_time`

Acquisition time. Use the ADC_ACQ_TIME macro to compose the value for this field or pass ADC_ACQ_TIME_DEFAULT to use the default setting for a given hardware (e.g. when the hardware does not allow to configure the acquisition time). Particular drivers do not necessarily support all the possible units. Value range is 0-16383 for a given unit.

uint8_t `channel_id`

Channel identifier. This value primarily identifies the channel within the ADC API - when a read request is done, the corresponding bit in the "channels" field of the "adc_sequence" structure must be set to include this channel in the sampling. For hardware that does not allow selection of analog inputs for given channels, but rather have dedicated ones, this value also selects the physical ADC input to be used in the sampling. Otherwise, when it is needed to explicitly select an analog input for the channel, or two inputs when the channel is a differential one, the selection is done in "input_positive" and "input_negative" fields. Particular drivers indicate which one of the above two cases they support by selecting or not a special hidden Kconfig option named ADC_CONFIGURABLE_INPUTS. If this option is not selected, the macro CONFIG_ADC_CONFIGURABLE_INPUTS is not defined and consequently the mentioned two fields are not present in this structure. While this API allows identifiers from range 0-31, particular drivers may support only a limited number of channel identifiers (dependent on the underlying hardware capabilities or configured via a dedicated Kconfig option).

uint8_t `differential`

Channel type: single-ended or differential.

struct `adc_sequence_options`

#include <adc.h> Structure defining additional options for an ADC sampling sequence.

Public Members

uint32_t `interval_us`

Interval between consecutive samplings (in microseconds), 0 means sample as fast as possible, without involving any timer. The accuracy of this interval is dependent on the

implementation of a given driver. The default routine that handles the intervals uses a kernel timer for this purpose, thus, it has the accuracy of the kernel's system clock. Particular drivers may use some dedicated hardware timers and achieve a better precision.

adc_sequence_callback `callback`

Callback function to be called after each sampling is done. Optional - set to NULL if it is not needed.

void *`user_data`

Pointer to user data. It can be used to associate the sequence with any other data that is needed in the callback function.

uint16_t `extra_samplings`

Number of extra samplings to perform (the total number of samplings is 1 + extra_samplings).

struct `adc_sequence`

#include <adc.h> Structure defining an ADC sampling sequence.

Public Members

const struct *adc_sequence_options* *`options`

Pointer to a structure defining additional options for the sequence. If NULL, the sequence consists of a single sampling.

uint32_t `channels`

Bit-mask indicating the channels to be included in each sampling of this sequence. All selected channels must be configured with *adc_channel_setup()* before they are used in a sequence.

void *`buffer`

Pointer to a buffer where the samples are to be written. Samples from subsequent samplings are written sequentially in the buffer. The number of samples written for each sampling is determined by the number of channels selected in the "channels" field. The buffer must be of an appropriate size, taking into account the number of selected channels and the ADC resolution used, as well as the number of samplings contained in the sequence.

size_t `buffer_size`

Specifies the actual size of the buffer pointed by the "buffer" field (in bytes). The driver must ensure that samples are not written beyond the limit and it must return an error if the buffer turns out to be not large enough to hold all the requested samples.

uint8_t `resolution`

ADC resolution. For single-ended channels the sample values are from range: 0 .. 2 ^ resolution - 1, for differential ones:

- 2 ^ (resolution-1) .. 2 ^ (resolution-1) - 1.

uint8_t `oversampling`

Oversampling setting. Each sample is averaged from 2 ^ oversampling conversion results. This feature may be unsupported by a given ADC hardware, or in a specific mode (e.g. when sampling multiple channels).

bool `calibrate`

Perform calibration before the reading is taken if requested.

The impact of channel configuration on the calibration process is specific to the underlying hardware. ADC implementations that do not support calibration should ignore this flag.

struct `adc_driver_api`

#include <adc.h> ADC driver API.

This is the mandatory API any ADC driver needs to expose.

7.21.2 Counter

Overview

API Reference

group `counter_interface`

Counter Interface.

Typedefs

typedef void (*`counter_alarm_callback_t`)(const struct *device* *dev, uint8_t chan_id, uint32_t ticks, void *user_data)

Alarm callback.

Param dev Pointer to the device structure for the driver instance.

Param chan_id Channel ID.

Param ticks Counter value that triggered the alarm.

Param user_data User data.

typedef void (*`counter_top_callback_t`)(const struct *device* *dev, void *user_data)

Callback called when counter turns around.

Param dev Pointer to the device structure for the driver instance.

Param user_data User data provided in *counter_set_top_value*.

typedef int (*`counter_api_start`)(const struct *device* *dev)

typedef int (*`counter_api_stop`)(const struct *device* *dev)

typedef int (*`counter_api_get_value`)(const struct *device* *dev, uint32_t *ticks)

typedef int (*`counter_api_set_alarm`)(const struct *device* *dev, uint8_t chan_id, const struct *counter_alarm_cfg* *alarm_cfg)

typedef int (*`counter_api_cancel_alarm`)(const struct *device* *dev, uint8_t chan_id)

typedef int (*counter_api_set_top_value)(const struct *device* *dev, const struct *counter_top_cfg* *cfg)

typedef uint32_t (*counter_api_get_pending_int)(const struct *device* *dev)

typedef uint32_t (*counter_api_get_top_value)(const struct *device* *dev)

typedef uint32_t (*counter_api_get_guard_period)(const struct *device* *dev, uint32_t flags)

typedef int (*counter_api_set_guard_period)(const struct *device* *dev, uint32_t ticks, uint32_t flags)

Functions

bool counter_is_counting_up(const struct *device* *dev)

Function to check if counter is counting up.

Parameters

- dev – **[in]** Pointer to the device structure for the driver instance.

Return values

- true – if counter is counting up.
- false – if counter is counting down.

uint8_t counter_get_num_of_channels(const struct *device* *dev)

Function to get number of alarm channels.

Parameters

- dev – **[in]** Pointer to the device structure for the driver instance.

Returns Number of alarm channels.

uint32_t counter_get_frequency(const struct *device* *dev)

Function to get counter frequency.

Parameters

- dev – **[in]** Pointer to the device structure for the driver instance.

Returns Frequency of the counter in Hz, or zero if the counter does not have a fixed frequency.

uint32_t counter_us_to_ticks(const struct *device* *dev, uint64_t us)

Function to convert microseconds to ticks.

Parameters

- dev – **[in]** Pointer to the device structure for the driver instance.
- us – **[in]** Microseconds.

Returns Converted ticks. Ticks will be saturated if exceed 32 bits.

uint64_t counter_ticks_to_us(const struct *device* *dev, uint32_t ticks)

Function to convert ticks to microseconds.

Parameters

- dev – **[in]** Pointer to the device structure for the driver instance.

- `ticks` – **[in]** Ticks.

Returns Converted microseconds.

uint32_t `counter_get_max_top_value`(const struct *device* *dev)

Function to retrieve maximum top value that can be set.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Returns Max top value.

int `counter_start`(const struct *device* *dev)

Start counter device in free running mode.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `counter_stop`(const struct *device* *dev)

Stop counter device.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `-ENOTSUP` – if the device doesn't support stopping the counter.

int `counter_get_value`(const struct *device* *dev, uint32_t *ticks)

Get current counter value.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ticks` – Pointer to where to store the current counter value

Return values

- `0` – If successful.
- `Negative` – error code on failure getting the counter value

int `counter_set_channel_alarm`(const struct *device* *dev, uint8_t chan_id, const struct *counter_alarm_cfg* *alarm_cfg)

Set a single shot alarm on a channel.

After expiration alarm can be set again, disabling is not needed. When alarm expiration handler is called, channel is considered available and can be set again in that context.

Note: API is not thread safe.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `chan_id` – Channel ID.
- `alarm_cfg` – Alarm configuration.

Return values

- `0` – If successful.
- `-ENOTSUP` – if request is not supported (device does not support interrupts or requested channel).
- `-EINVAL` – if alarm settings are invalid.
- `-ETIME` – if absolute alarm was set too late.
- `-EBUSY` – if alarm is already active.

int `counter_cancel_channel_alarm`(const struct *device* *dev, uint8_t chan_id)

Cancel an alarm on a channel.

Note: API is not thread safe.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `chan_id` – Channel ID.

Return values

- `0` – If successful.
- `-ENOTSUP` – if request is not supported or the counter was not started yet.

int `counter_set_top_value`(const struct *device* *dev, const struct *counter_top_cfg* *cfg)

Set counter top value.

Function sets top value and optionally resets the counter to 0 or top value depending on counter direction. On turnaround, counter can be reset and optional callback is periodically called. Top value can only be changed when there is no active channel alarm.

COUNTER_TOP_CFG_DONT_RESET prevents counter reset. When counter is running while top value is updated, it is possible that counter progresses outside the new top value. In that case, error is returned and optionally driver can reset the counter (see COUNTER_TOP_CFG_RESET_WHEN_LATE).

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cfg` – Configuration. Cannot be NULL.

Return values

- `0` – If successful.
- `-ENOTSUP` – if request is not supported (e.g. top value cannot be changed or counter cannot/must be reset during top value update).
- `-EBUSY` – if any alarm is active.
- `-ETIME` – if COUNTER_TOP_CFG_DONT_RESET was set and new top value is smaller than current counter value (counter counting up).

int `counter_get_pending_int`(const struct *device* *dev)

Function to get pending interrupts.

The purpose of this function is to return the interrupt status register for the device. This is especially useful when waking up from low power states to check the wake up source.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `1` – if any counter interrupt is pending.
- `0` – if no counter interrupt is pending.

uint32_t `counter_get_top_value`(const struct *device* *dev)

Function to retrieve current top value.

Parameters

- `dev` – **[in]** Pointer to the device structure for the driver instance.

Returns Top value.

int `counter_set_guard_period`(const struct *device* *dev, uint32_t ticks, uint32_t flags)

Set guard period in counter ticks.

Setting non-zero guard period enables detection of setting absolute alarm too late. It limits how far in the future absolute alarm can be set.

Detection of too late setting is vital since if it is not detected alarm is delayed by full period of the counter (up to 32 bits). Because of the wrapping, it is impossible to distinguish alarm which is short in the past from alarm which is targeted to expire after full counter period. In order to detect too late setting, longest possible alarm is limited. Absolute value cannot exceed: (now + top_value - guard_period) % top_value.

Guard period depends on application and counter frequency. If it is expected that absolute alarms setting might be delayed then guard period should exceed maximal potential delay. If use case allows, guard period can be set very high (e.g. half of the counter top value).

After initialization guard period is set to 0 and late detection is disabled.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ticks` – Guard period in counter ticks.
- `flags` – See Counter guard period flags.

Return values

- `0` – if successful.
- `-ENOTSUP` – if function or flags are not supported.
- `-EINVAL` – if ticks value is invalid.

uint32_t `counter_get_guard_period`(const struct *device* *dev, uint32_t flags)

Return guard period.

See *counter_set_guard_period*.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `flags` – See Counter guard period flags.

Returns Guard period given in counter ticks or 0 if function or flags are not supported.

struct `counter_alarm_cfg`

#include <counter.h> Alarm callback structure.

Param callback Callback called on alarm (cannot be NULL).

Param ticks Number of ticks that triggers the alarm. It can be relative (to now) or absolute value (see COUNTER_ALARM_CFG_ABSOLUTE). Absolute alarm cannot be set further in future than top_value decremented by the guard period. Relative alarm ticks cannot exceed current top value (see *counter_get_top_value*). If counter is clock driven then ticks can be converted to microseconds (see *counter_ticks_to_us*). Alternatively, counter implementation may count asynchronous events.

Param user_data User data returned in callback.

Param flags Alarm flags. See Alarm configuration flags.

struct `counter_top_cfg`

#include <counter.h> Top value configuration structure.

Param ticks Top value.

Param callback Callback function. Can be NULL.

Param user_data User data passed to callback function. Not valid if callback is NULL.

Param flags Flags. See Flags used by .

struct `counter_config_info`

#include <counter.h> Structure with generic counter features.

Param max_top_value Maximal (default) top value on which counter is reset (cleared or reloaded).

Param freq Frequency of the source clock if synchronous events are counted.

Param flags Flags. See Counter device capabilities.

Param channels Number of channels that can be used for setting alarm, see *counter_set_channel_alarm*.

struct `counter_driver_api`

#include <counter.h>

7.21.3 Clock Control

Overview

The clock control API provides access to clocks in the system, including the ability to turn them on and off.

Configuration Options

Related configuration options:

- `CONFIG_CLOCK_CONTROL`

API Reference

group `clock_control_interface`

Clock Control Interface.

Defines

CLOCK_CONTROL_SUBSYS_ALL

Typedefs

typedef void *clock_control_subsys_t

clock_control_subsys_t is a type to identify a clock controller sub-system. Such data pointed is opaque and relevant only to the clock controller driver instance being used.

typedef void (*clock_control_cb_t)(const struct *device* *dev, *clock_control_subsys_t* subsys, void *user_data)

Callback called on clock started.

Param dev Device structure whose driver controls the clock.

Param subsys Opaque data representing the clock.

Param user_data User data.

typedef int (*clock_control)(const struct *device* *dev, *clock_control_subsys_t* sys)

typedef int (*clock_control_get)(const struct *device* *dev, *clock_control_subsys_t* sys, uint32_t *rate)

typedef int (*clock_control_async_on_fn)(const struct *device* *dev, *clock_control_subsys_t* sys, *clock_control_cb_t* cb, void *user_data)

typedef enum *clock_control_status* (*clock_control_get_status_fn)(const struct *device* *dev, *clock_control_subsys_t* sys)

Enums

enum clock_control_status

Current clock status.

Values:

enumerator CLOCK_CONTROL_STATUS_STARTING

enumerator CLOCK_CONTROL_STATUS_OFF

enumerator CLOCK_CONTROL_STATUS_ON

enumerator CLOCK_CONTROL_STATUS_UNAVAILABLE

enumerator CLOCK_CONTROL_STATUS_UNKNOWN

Functions

static inline int `clock_control_on`(const struct *device* *dev, *clock_control_subsys_t* sys)

Enable a clock controlled by the device.

On success, the clock is enabled and ready when this function returns. This function may sleep, and thus can only be called from thread context.

Use *clock_control_async_on()* for non-blocking operation.

Parameters

- `dev` – Device structure whose driver controls the clock.
- `sys` – Opaque data representing the clock.

Returns 0 on success, negative errno on failure.

static inline int `clock_control_off`(const struct *device* *dev, *clock_control_subsys_t* sys)

Disable a clock controlled by the device.

This function is non-blocking and can be called from any context. On success, the clock is disabled when this function returns.

Parameters

- `dev` – Device structure whose driver controls the clock
- `sys` – Opaque data representing the clock

Returns 0 on success, negative errno on failure.

static inline int `clock_control_async_on`(const struct *device* *dev, *clock_control_subsys_t* sys, *clock_control_cb_t* cb, void *user_data)

Request clock to start with notification when clock has been started.

Function is non-blocking and can be called from any context. User callback is called when clock is started.

Parameters

- `dev` – Device.
- `sys` – A pointer to an opaque data representing the sub-system.
- `cb` – Callback.
- `user_data` – User context passed to the callback.

Return values

- `0` – if start is successfully initiated.
- `-EALREADY` – if clock was already started and is starting or running.
- `-ENOTSUP` – If the requested mode of operation is not supported.
- `-ENOSYS` – if the interface is not implemented.
- `other` – negative errno on vendor specific error.

static inline enum *clock_control_status* `clock_control_get_status`(const struct *device* *dev, *clock_control_subsys_t* sys)

Get clock status.

Parameters

- `dev` – Device.
- `sys` – A pointer to an opaque data representing the sub-system.

Returns Status.

static inline int clock_control_get_rate(const struct *device* *dev, *clock_control_subsys_t* sys, uint32_t *rate)

Obtain the clock rate of given sub-system.

Parameters

- dev – Pointer to the device structure for the clock controller driver instance
- sys – A pointer to an opaque data representing the sub-system
- rate – **[out]** Subsystem clock rate

struct clock_control_driver_api

#include <clock_control.h>

7.21.4 DAC

Overview

The DAC API provides access to Digital-to-Analog Converter (DAC) devices.

Configuration Options

Related configuration options:

- CONFIG_DAC

API Reference

group dac_interface

DAC driver APIs.

Functions

int dac_channel_setup(const struct *device* *dev, const struct *dac_channel_cfg* *channel_cfg)

Configure a DAC channel.

It is required to call this function and configure each channel before it is selected for a write request.

Parameters

- dev – Pointer to the device structure for the driver instance.
- channel_cfg – Channel configuration.

Return values

- 0 – On success.
- -EINVAL – If a parameter with an invalid value has been provided.
- -ENOTSUP – If the requested resolution is not supported.

int dac_write_value(const struct *device* *dev, uint8_t channel, uint32_t value)

Write a single value to a DAC channel.

Parameters

- dev – Pointer to the device structure for the driver instance.

- `channel` – Number of the channel to be used.
- `value` – Data to be written to DAC output registers.

Return values

- `0` – On success.
- `-EINVAL` – If a parameter with an invalid value has been provided.

struct `dac_channel_cfg`

#include <dac.h> Structure for specifying the configuration of a DAC channel.

Param channel_id Channel identifier of the DAC that should be configured.

Param resolution Desired resolution of the DAC (depends on device capabilities).

7.21.5 DMA

Overview

API Reference

group `dma_interface`

DMA Interface.

Defines

`DMA_MAGIC`

Typedefs

typedef void (*`dma_callback_t`)(const struct *device* *dev, void *user_data, uint32_t channel, int status)

Callback function for DMA transfer completion.

If enabled, callback function will be invoked at transfer completion or when error happens.

Param dev Pointer to the DMA device calling the callback.

Param user_data A pointer to some user data or NULL

Param channel The channel number

Param status 0 on success, a negative errno otherwise

Enums

enum `dma_channel_direction`

Values:

enumerator `MEMORY_TO_MEMORY` = 0x0

enumerator MEMORY_TO_PERIPHERAL

enumerator PERIPHERAL_TO_MEMORY

enumerator PERIPHERAL_TO_PERIPHERAL

enum dma_addr_adj

Valid values for *source_addr_adj* and *dest_addr_adj*

Values:

enumerator DMA_ADDR_ADJ_INCREMENT

enumerator DMA_ADDR_ADJ_DECREMENT

enumerator DMA_ADDR_ADJ_NO_CHANGE

enum dma_channel_filter

Values:

enumerator DMA_CHANNEL_NORMAL

enumerator DMA_CHANNEL_PERIODIC

Functions

static inline int dma_config(const struct *device* *dev, uint32_t channel, struct *dma_config* *config)

Configure individual channel for DMA transfer.

Parameters

- dev – Pointer to the device structure for the driver instance.
- channel – Numeric identification of the channel to configure
- config – Data structure containing the intended configuration for the selected channel

Return values

- 0 – if successful.
- Negative – errno code if failure.

static inline int dma_reload(const struct *device* *dev, uint32_t channel, uint32_t src, uint32_t dst, size_t size)

Reload buffer(s) for a DMA channel.

Parameters

- dev – Pointer to the device structure for the driver instance.
- channel – Numeric identification of the channel to configure selected channel
- src – source address for the DMA transfer
- dst – destination address for the DMA transfer

- `size` – size of DMA transfer

Return values

- `0` – if successful.
- `Negative` – errno code if failure.

int `dma_start`(const struct *device* *dev, uint32_t channel)

Enables DMA channel and starts the transfer, the channel must be configured beforehand.

Implementations must check the validity of the channel ID passed in and return -EINVAL if it is invalid.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – Numeric identification of the channel where the transfer will be processed

Return values

- `0` – if successful.
- `Negative` – errno code if failure.

int `dma_stop`(const struct *device* *dev, uint32_t channel)

Stops the DMA transfer and disables the channel.

Implementations must check the validity of the channel ID passed in and return -EINVAL if it is invalid.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – Numeric identification of the channel where the transfer was being processed

Return values

- `0` – if successful.
- `Negative` – errno code if failure.

int `dma_request_channel`(const struct *device* *dev, void *filter_param)

request DMA channel.

request DMA channel resources return -EINVAL if there is no valid channel available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `filter_param` – filter function parameter

Return values

- `dma` – channel if successful.
- `Negative` – errno code if failure.

void `dma_release_channel`(const struct *device* *dev, uint32_t channel)

release DMA channel.

release DMA channel resources

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – channel number

int `dma_chan_filter`(const struct *device* *dev, int channel, void *filter_param)

DMA channel filter.

filter channel by attribute

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – channel number
- `filter_param` – filter attribute

Return values `Negative` – errno code if not support

static inline int `dma_get_status`(const struct *device* *dev, uint32_t channel, struct *dma_status* *stat)

get current runtime status of DMA transfer

Implementations must check the validity of the channel ID passed in and return -EINVAL if it is invalid or -ENOSYS if not supported.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `channel` – Numeric identification of the channel where the transfer was being processed
- `stat` – a non-NULL *dma_status* object for storing DMA status

Return values

- `non-negative` – if successful.
- `Negative` – errno code if failure.

static inline uint32_t `dma_width_index`(uint32_t size)

Look-up generic width index to be used in registers.

WARNING: This look-up works for most controllers, but *may* not work for yours. Ensure your controller expects the most common register bit values before using this convenience function. If your controller does not support these values, you will have to write your own look-up inside the controller driver.

Parameters

- `size` – width of bus (in bytes)

Return values `common` – DMA index to be placed into registers.

static inline uint32_t `dma_burst_index`(uint32_t burst)

Look-up generic burst index to be used in registers.

WARNING: This look-up works for most controllers, but *may* not work for yours. Ensure your controller expects the most common register bit values before using this convenience function. If your controller does not support these values, you will have to write your own look-up inside the controller driver.

Parameters

- `burst` – number of bytes to be sent in a single burst

Return values `common` – DMA index to be placed into registers.

struct `dma_block_config`

#include <dma.h> DMA block configuration structure.

Param source_address is block starting address at source

Param source_gather_interval is the address adjustment at gather boundary

Param dest_address is block starting address at destination

Param dest_scatter_interval is the address adjustment at scatter boundary

Param dest_scatter_count is the continuous transfer count between scatter boundaries

Param source_gather_count is the continuous transfer count between gather boundaries

Param block_size is the number of bytes to be transferred for this block.

Param config is a bit field with the following parts:

```
source_gather_en    [ 0 ]       - 0-disable, 1-enable.
dest_scatter_en     [ 1 ]       - 0-disable, 1-enable.
source_addr_adj     [ 2 : 3 ]   - 00-increment, 01-decrement,
                                  10-no change.
dest_addr_adj       [ 4 : 5 ]   - 00-increment, 01-decrement,
                                  10-no change.
source_reload_en    [ 6 ]       - reload source address at the end␣
↪of
                                  block transfer
                                  0-disable, 1-enable.
dest_reload_en      [ 7 ]       - reload destination address at␣
↪the end
                                  of block transfer
                                  0-disable, 1-enable.
fifo_mode_control   [ 8 : 11 ]  - How full  of the fifo before␣
↪transfer
                                  start. HW specific.
flow_control_mode   [ 12 ]      - 0-source request served upon data
                                    availability.
                                  1-source request postponed until
                                    destination request happens.
reserved            [ 13 : 15 ]
```

struct `dma_config`

#include <dma.h> DMA configuration structure.

Param dma_slot [0 : 6] - which peripheral and direction (HW specific)

Param channel_direction [7 : 9] - 000-memory to memory, 001-memory to peripheral, 010-peripheral to memory, 011-peripheral to peripheral, . . .

Param complete_callback_en [10] - 0-callback invoked at completion only 1-callback invoked at completion of each block

Param error_callback_en [11] - 0-error callback enabled 1-error callback disabled

Param source_handshake [12] - 0-HW, 1-SW

Param dest_handshake [13] - 0-HW, 1-SW

Param channel_priority [14 : 17] - DMA channel priority

Param source_chaining_en [18] - enable/disable source block chaining 0-disable, 1-enable

Param dest_chaining_en [19] - enable/disable destination block chaining. 0-disable, 1-enable

Param linked_channel [20 : 26] - after channel count exhaust will initiate a channel service request at this channel

Param reserved [27 : 31]

Param source_data_size [0 : 15] - width of source data (in bytes)

Param dest_data_size [16 : 31] - width of dest data (in bytes)

Param source_burst_length [0 : 15] - number of source data units

Param dest_burst_length [16 : 31] - number of destination data units

Param block_count is the number of blocks used for block chaining, this depends on availability of the DMA controller.

Param user_data private data from DMA client.

Param dma_callback see dma_callback_t for details

struct `dma_status`

#include <dma.h> DMA runtime status structure

busy - is current DMA transfer busy or idle dir - DMA transfer direction pending_length - data length pending to be transferred in bytes or platform dependent.

struct `dma_context`

#include <dma.h> DMA context structure Note: the *dma_context* shall be the first member of DMA client driver Data, got by dev->data

magic - magic code to identify the context dma_channels - dma channels atomic - driver atomic_t pointer

7.21.6 EC Host Command

Overview

API Reference

group `ec_host_cmd_periph_interface`

EC Host Command Interface.

Defines

`EC_HOST_CMD_HANDLER(_function, _id, _version_mask, _request_type, _response_type)`

Statically define and register a host command handler.

Helper macro to statically define and register a host command handler that has a compile-time-fixed sizes for its both request and response structures.

Parameters

- `_function` – Name of handler function.
- `_id` – Id of host command to handle request for.
- `_version_mask` – The bitfield of all versions that the *_function* supports. E.g. *BIT(0)* corresponse to version 0.
- `_request_type` – The datatype of the request parameters for *_function*.
- `_response_type` – The datatype of the response parameters for *_function*.

EC_HOST_CMD_HANDLER_UNBOUND(_function, _id, _version_mask)

Statically define and register a host command handler without sizes.

Helper macro to statically define and register a host command handler whose request or response structure size is not known as compile time.

Parameters

- `_function` – Name of handler function.
- `_id` – Id of host command to handle request for.
- `_version_mask` – The bitfield of all versions that the *_function* supports. E.g. *BIT(0)* corresponse to version 0.

Typedefs

typedef enum *ec_host_cmd_status* (*`ec_host_cmd_handler_cb`)(struct *ec_host_cmd_handler_args* *args)

Enums

enum `ec_host_cmd_status`

Values:

enumerator `EC_HOST_CMD_SUCCESS` = 0

Host command was successful.

enumerator `EC_HOST_CMD_INVALID_COMMAND` = 1

The specified command id is not recognized or supported.

enumerator `EC_HOST_CMD_ERROR` = 2

Generic Error.

enumerator `EC_HOST_CMD_INVALID_PARAM` = 3

One of more of the input request parameters is invalid.

enumerator `EC_HOST_CMD_ACCESS_DENIED` = 4

Host command is not permitted.

enumerator `EC_HOST_CMD_INVALID_RESPONSE` = 5

Response was invalid (e.g. not version 3 of header).

enumerator `EC_HOST_CMD_INVALID_VERSION` = 6

Host command id version unsupported.

enumerator `EC_HOST_CMD_INVALID_CHECKSUM` = 7

Checksum did not match

enumerator `EC_HOST_CMD_IN_PROGRESS` = 8

A host command is currently being processed.

enumerator EC_HOST_CMD_UNAVAILABLE = 9

Requested information is currently unavailable.

enumerator EC_HOST_CMD_TIMEOUT = 10

Timeout during processing.

enumerator EC_HOST_CMD_OVERFLOW = 11

Data or table overflow.

enumerator EC_HOST_CMD_INVALID_HEADER = 12

Header is invalid or unsupported (e.g. not version 3 of header).

enumerator EC_HOST_CMD_REQUEST_TRUNCATED = 13

Did not receive all expected request data.

enumerator EC_HOST_CMD_RESPONSE_TOO_BIG = 14

Response was too big to send within one response packet.

enumerator EC_HOST_CMD_BUS_ERROR = 15

Error on underlying communication bus.

enumerator EC_HOST_CMD_BUSY = 16

System busy. Should retry later.

enumerator EC_HOST_CMD_MAX = UINT16_MAX

struct ec_host_cmd_handler_args

#include <ec_host_cmd.h> Arguments passed into every installed host command handler.

Public Members

const void *const input_buf

The incoming data that can be cast to the handlers request type.

const uint16_t input_buf_size

The number of valid bytes that can be read from *input_buf*.

void *const output_buf

The data written to this buffer will be send to the host.

uint16_t output_buf_size

[in/out] Upon entry, this is the maximum number of bytes that can be written to the *output_buf*. Upon exit, this should be the number of bytes of *output_buf* to send to the host.

const uint8_t version

The version of the host command that is being requested. This will be a value that has been static registered as valid for the handler.

struct ec_host_cmd_handler

#include <ec_host_cmd.h> Structure use for statically registering host command handlers.

Public Members

ec_host_cmd_handler_cb handler

Callback routine to process commands that match *id*.

uint16_t id

The numberical command id used as the lookup for commands.

uint16_t version_mask

The bitfield of all versions that the *handler* supports, where each bit value represents that the *handler* supports that version. E.g. *BIT(0)* corresponse to version 0.

uint16_t min_rqt_size

The minimum *input_buf_size* enforced by the framework before passing to the handler.

uint16_t min_rsp_size

The minimum *output_buf_size* enforced by the framework before passing to the handler.

struct ec_host_cmd_request_header

#include <ec_host_cmd.h> Header for requests from host to embedded controller.

Represent the over-the-wire header in LE format for host command requests. This represent version 3 of the host command header. The requests are always sent from host to embedded controller.

Public Members

uint8_t prtcl_ver

Should be 3. The EC will return EC_HOST_CMD_INVALID_HEADER if it receives a header with a version it doesn't know how to parse.

uint8_t checksum

Checksum of response and data; sum of all bytes including checksum. Should total to 0.

uint16_t cmd_id

Id of command that is being sent.

uint8_t cmd_ver

Version of the specific *cmd_id* being requested. Valid versions start at 0.

uint8_t reserved

Unused byte in current protocol version; set to 0.

uint16_t data_len

Length of data which follows this header.

struct `ec_host_cmd_response_header`

#include <ec_host_cmd.h> Header for responses from embedded controller to host.

Represent the over-the-wire header in LE format for host command responses. This represent version 3 of the host command header. Responses are always sent from embedded controller to host.

Public Members

uint8_t `prtcl_ver`

Should be 3.

uint8_t `checksum`

Checksum of response and data; sum of all bytes including checksum. Should total to 0.

uint16_t `result`

A *ec_host_cmd_status* response code for specific command.

uint16_t `data_len`

Length of data which follows this header.

uint16_t `reserved`

Unused bytes in current protocol version; set to 0.

7.21.7 EEPROM

Overview

The EEPROM API provides read and write access to Electrically Erasable Programmable Read-Only Memory (EEPROM) devices.

EEPROMs have an erase block size of 1 byte, a long lifetime, and allow overwriting data on byte-by-byte access.

Configuration Options

Related configuration options:

- `CONFIG_EEPROM`

API Reference

group `eeprom_interface`

EEPROM Interface.

Typedefs

typedef int (*`eeprom_api_read`)(const struct *device* *dev, off_t offset, void *data, size_t len)

typedef int (*eeprom_api_write)(const struct *device* *dev, off_t offset, const void *data, size_t len)

typedef size_t (*eeprom_api_size)(const struct *device* *dev)

Functions

int eeprom_read(const struct *device* *dev, off_t offset, void *data, size_t len)

Read data from EEPROM.

Parameters

- dev – EEPROM device
- offset – Address offset to read from.
- data – Buffer to store read data.
- len – Number of bytes to read.

Returns 0 on success, negative errno code on failure.

int eeprom_write(const struct *device* *dev, off_t offset, const void *data, size_t len)

Write data to EEPROM.

Parameters

- dev – EEPROM device
- offset – Address offset to write data to.
- data – Buffer with data to write.
- len – Number of bytes to write.

Returns 0 on success, negative errno code on failure.

size_t eeprom_get_size(const struct *device* *dev)

Get the size of the EEPROM in bytes.

Parameters

- dev – EEPROM device.

Returns EEPROM size in bytes.

struct eeprom_driver_api

#include <eeprom.h>

7.21.8 Entropy

Overview

The entropy API provides functions to retrieve entropy values from entropy hardware present on the platform. The entropy APIs are provided for use by the random subsystem and cryptographic services. They are not suitable to be used as random number generation functions.

API Reference

group entropy_interface

Entropy Interface.

Defines

ENTROPY_BUSYWAIT

Typedefs

typedef int (*entropy_get_entropy_t)(const struct *device* *dev, uint8_t *buffer, uint16_t length)

Callback API to get entropy.

See *entropy_get_entropy()* for argument description

typedef int (*entropy_get_entropy_isr_t)(const struct *device* *dev, uint8_t *buffer, uint16_t length, uint32_t flags)

Callback API to get entropy from an ISR.

See *entropy_get_entropy_isr()* for argument description

Functions

int entropy_get_entropy(const struct *device* *dev, uint8_t *buffer, uint16_t length)

Fills a buffer with entropy. Blocks if required in order to generate the necessary random data.

Parameters

- dev – Pointer to the entropy device.
- buffer – Buffer to fill with entropy.
- length – Buffer length.

Return values

- 0 – on success.
- -ERRNO – errno code on error.

static inline int entropy_get_entropy_isr(const struct *device* *dev, uint8_t *buffer, uint16_t length, uint32_t flags)

Fills a buffer with entropy in a non-blocking or busy-wait manner. Callable from ISRs.

Parameters

- dev – Pointer to the device structure.
- buffer – Buffer to fill with entropy.
- length – Buffer length.
- flags – Flags to modify the behavior of the call.

Return values number – of bytes filled with entropy or -error.

struct entropy_driver_api

#include <entropy.h>

7.21.9 Flash

Overview

Flash offset concept

Offsets used by the user API are expressed in relation to the flash memory beginning address. This rule shall be applied to all flash controller regular memory that layout is accessible via API for retrieving the layout of pages (see option:`CONFIG_FLASH_PAGE_LAYOUT`).

An exception from the rule may be applied to a vendor-specific flash dedicated-purpose region (such a region obviously can't be covered under API for retrieving the layout of pages).

User API Reference

group `flash_interface`

FLASH Interface.

Typedefs

typedef bool (*`flash_page_cb`)(const struct *flash_pages_info* *info, void *data)

Callback type for iterating over flash pages present on a device.

The callback should return true to continue iterating, and false to halt.

See also:

flash_page_foreach()

Param info Information for current page

Param data Private data for callback

Return True to continue iteration, false to halt iteration.

Functions

int `flash_read`(const struct *device* *dev, off_t offset, void *data, size_t len)

Read data from flash.

All flash drivers support reads without alignment restrictions on the read offset, the read size, or the destination address.

Parameters

- `dev` – : flash dev
- `offset` – : Offset (byte aligned) to read
- `data` – : Buffer to store read data
- `len` – : Number of bytes to read.

Returns 0 on success, negative errno code on fail.

int `flash_write`(const struct *device* *dev, off_t offset, const void *data, size_t len)

Write buffer into flash memory.

All flash drivers support a source buffer located either in RAM or SoC flash, without alignment restrictions on the source address. Write size and offset must be multiples of the minimum write block size supported by the driver.

Any necessary write protection management is performed by the driver write implementation itself.

Parameters

- `dev` – : flash device
- `offset` – : starting offset for the write
- `data` – : data to write
- `len` – : Number of bytes to write

Returns 0 on success, negative errno code on fail.

int `flash_erase`(const struct *device* *dev, off_t offset, size_t size)

Erase part or all of a flash memory.

Acceptable values of erase size and offset are subject to hardware-specific multiples of page size and offset. Please check the API implemented by the underlying sub driver, for example by using *flash_get_page_info_by_offs()* if that is supported by your flash driver.

Any necessary erase protection management is performed by the driver erase implementation itself.

See also:

flash_get_page_info_by_offs()

See also:

flash_get_page_info_by_idx()

Parameters

- `dev` – : flash device
- `offset` – : erase area starting offset
- `size` – : size of area to be erased

Returns 0 on success, negative errno code on fail.

int `flash_write_protection_set`(const struct *device* *dev, bool enable)

Enable or disable write protection for a flash memory.

This API is deprecated and will be removed in Zephyr 2.8. It will be keep as No-Operation until removal. Flash write/erase protection management has been moved to write and erase operations implementations in flash driver shims. For Out-of-tree drivers which are not updated yet flash write/erase protection management is done in *flash_erase()* and *flash_write()* using deprecated

write_protection

shim handler.

Parameters

- `dev` – : flash device
- `enable` – : enable or disable flash write protection

Returns 0 on success, negative errno code on fail.

int `flash_get_page_info_by_offs`(const struct *device* *dev, off_t offset, struct *flash_pages_info* *info)

Get the size and start offset of flash page at certain flash offset.

Parameters

- `dev` – flash device
- `offset` – Offset within the page
- `info` – Page Info structure to be filled

Returns 0 on success, -EINVAL if page of the offset doesn't exist.

int `flash_get_page_info_by_idx`(const struct *device* *dev, uint32_t page_index, struct *flash_pages_info* *info)

Get the size and start offset of flash page of certain index.

Parameters

- `dev` – flash device
- `page_index` – Index of the page. Index are counted from 0.
- `info` – Page Info structure to be filled

Returns 0 on success, -EINVAL if page of the index doesn't exist.

size_t `flash_get_page_count`(const struct *device* *dev)

Get the total number of flash pages.

Parameters

- `dev` – flash device

Returns Number of flash pages.

void `flash_page_foreach`(const struct *device* *dev, *flash_page_cb* cb, void *data)

Iterate over all flash pages on a device.

This routine iterates over all flash pages on the given device, ordered by increasing start offset. For each page, it invokes the given callback, passing it the page's information and a private data object.

Parameters

- `dev` – Device whose pages to iterate over
- `cb` – Callback to invoke for each flash page
- `data` – Private data for callback function

int `flash_sfdp_read`(const struct *device* *dev, off_t offset, void *data, size_t len)

Read data from Serial Flash Discoverable Parameters.

This routine reads data from a serial flash device compatible with the JEDEC JESD216 standard for encoding flash memory characteristics.

Availability of this API is conditional on selecting `CONFIG_FLASH_JESD216_API` and support of that functionality in the driver underlying `dev`.

Parameters

- `dev` – device from which parameters will be read
- `offset` – address within the SFDP region containing data of interest
- `data` – where the data to be read will be placed
- `len` – the number of bytes of data to be read

Return values

- `0` – on success
- `-ENOTSUP` – if the flash driver does not support SFDP access
- `negative` – values for other errors.

int `flash_read_jedec_id`(const struct *device* *dev, uint8_t *id)

Read the JEDEC ID from a compatible flash device.

Parameters

- `dev` – device from which id will be read
- `id` – pointer to a buffer of at least 3 bytes into which id will be stored

Return values

- `0` – on successful store of 3-byte JEDEC id
- `-ENOTSUP` – if flash driver doesn't support this function
- `negative` – values for other errors

size_t `flash_get_write_block_size`(const struct *device* *dev)

Get the minimum write block size supported by the driver.

The write block size supported by the driver might differ from the write block size of memory used because the driver might implements write-modify algorithm.

Parameters

- `dev` – flash device

Returns write block size in bytes.

const struct *flash_parameters* *`flash_get_parameters`(const struct *device* *dev)

Get pointer to *flash_parameters* structure.

Returned pointer points to a structure that should be considered constant through a runtime, regardless if it is defined in RAM or Flash. Developer is free to cache the structure pointer or copy its contents.

Returns pointer to *flash_parameters* structure characteristic for the device.

struct `flash_parameters`

#include <flash.h> Flash memory parameters. Contents of this structure suppose to be filled in during flash device initialization and stay constant through a runtime.

struct `flash_pages_info`

#include <flash.h>

Implementation interface API Reference

group `flash_internal_interface`

FLASH internal Interface.

Typedefs

typedef int (*`flash_api_read`)(const struct *device* *dev, off_t offset, void *data, size_t len)

typedef int (*flash_api_write)(const struct *device* *dev, off_t offset, const void *data, size_t len)

Flash write implementation handler type.

Note: Any necessary write protection management must be performed by the driver, with the driver responsible for ensuring the "write-protect" after the operation completes (successfully or not) matches the write-protect state when the operation was started.

typedef int (*flash_api_erase)(const struct *device* *dev, off_t offset, size_t size)

Flash erase implementation handler type.

Note: Any necessary erase protection management must be performed by the driver, with the driver responsible for ensuring the "erase-protect" after the operation completes (successfully or not) matches the erase-protect state when the operation was started.

typedef int (*flash_api_write_protection)(const struct *device* *dev, bool enable)

typedef const struct *flash_parameters* *(*flash_api_get_parameters)(const struct *device* *dev)

typedef void (*flash_api_pages_layout)(const struct *device* *dev, const struct *flash_pages_layout* **layout, size_t *layout_size)

Retrieve a flash device's layout.

A flash device layout is a run-length encoded description of the pages on the device. (Here, "page" means the smallest erasable area on the flash device.)

For flash memories which have uniform page sizes, this routine returns an array of length 1, which specifies the page size and number of pages in the memory.

Layouts for flash memories with nonuniform page sizes will be returned as an array with multiple elements, each of which describes a group of pages that all have the same size. In this case, the sequence of array elements specifies the order in which these groups occur on the device.

Param dev Flash device whose layout to retrieve.

Param layout The flash layout will be returned in this argument.

Param layout_size The number of elements in the returned layout.

typedef int (*flash_api_sfdp_read)(const struct *device* *dev, off_t offset, void *data, size_t len)

typedef int (*flash_api_read_jedec_id)(const struct *device* *dev, uint8_t *id)

struct flash_pages_layout

#include <flash.h>

struct flash_driver_api

#include <flash.h>

7.21.10 GNA

Overview

The GNA API provides access to Intel's Gaussian Mixture Model and Neural Network Accelerator (GNA).

Configuration Options

Related configuration options:

- `CONFIG_INTEL_GNA`

API Reference

group `gna_interface`

This file contains the driver APIs for Intel's Gaussian Mixture Model and Neural Network Accelerator (GNA)

Enums

enum `gna_result`

Result of an inference operation

Values:

enumerator `GNA_RESULT_INFERENCE_COMPLETE`

enumerator `GNA_RESULT_SATURATION_OCCURRED`

enumerator `GNA_RESULT_OUTPUT_BUFFER_FULL_ERROR`

enumerator `GNA_RESULT_PARAM_OUT_OF_RANGE_ERROR`

enumerator `GNA_RESULT_GENERIC_ERROR`

Functions

static inline int `gna_configure`(const struct *device* *dev, struct *gna_config* *cfg)

Configure the GNA device.

Configure the GNA device. The GNA device must be configured before registering a model or performing inference

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cfg` – Device configuration information

Return values

- `0` – If the configuration is successful
- `A` – negative error code in case of a failure.

static inline int gna_register_model(const struct *device* *dev, struct *gna_model_info* *model, void **model_handle)

Register a neural network model.

Register a neural network model with the GNA device A model needs to be registered before it can be used to perform inference

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `model` – Information about the neural network model
- `model_handle` – Handle to the registered model if registration succeeds

Return values

- `0` – If registration of the model is successful.
- `A` – negative error code in case of a failure.

static inline int gna_deregister_model(const struct *device* *dev, void *model)

De-register a previously registered neural network model.

De-register a previously registered neural network model from the GNA device De-registration may be done to free up memory for registering another model Once de-registered, the model can no longer be used to perform inference

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `model` – Model handle output by gna_register_model API

Return values

- `0` – If de-registration of the model is successful.
- `A` – negative error code in case of a failure.

static inline int gna_infer(const struct *device* *dev, struct *gna_inference_req* *req, gna_callback callback)

Perform inference on a model with input vectors.

Make an inference request on a previously registered model with an of input data vector A callback is provided for notification of inference completion

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `req` – Information required to perform inference on a neural network
- `callback` – A callback function to notify inference completion

Return values

- `0` – If the request is accepted
- `A` – negative error code in case of a failure.

struct gna_config

#include <gna.h> GNA driver configuration structure. Currently empty.

struct gna_model_header

#include <gna.h> GNA Neural Network model header Describes the key parameters of the neural network model

struct gna_model_info

#include <gna.h> GNA Neural Network model information to be provided by application during model registration

struct gna_inference_req

#include <gna.h> Request to perform inference on the given neural network model

struct gna_inference_stats

#include <gna.h> Statistics of the inference operation returned after completion

struct gna_inference_resp

#include <gna.h> Structure containing a response to the inference request

7.21.11 GPIO

Overview

Configuration Options

Related configuration options:

- CONFIG_GPIO

API Reference

group gpio_interface

GPIO Driver APIs.

GPIO input/output configuration flags

GPIO_INPUT

Enables pin as input.

GPIO_OUTPUT

Enables pin as output, no change to the output state.

GPIO_DISCONNECTED

Disables pin for both input and output.

GPIO_OUTPUT_LOW

Configures GPIO pin as output and initializes it to a low state.

GPIO_OUTPUT_HIGH

Configures GPIO pin as output and initializes it to a high state.

GPIO_OUTPUT_INACTIVE

Configures GPIO pin as output and initializes it to a logic 0.

GPIO_OUTPUT_ACTIVE

Configures GPIO pin as output and initializes it to a logic 1.

GPIO interrupt configuration flags

The GPIO_INT_* flags are used to specify how input GPIO pins will trigger interrupts. The interrupts can be sensitive to pin physical or logical level. Interrupts sensitive to pin logical level take into account GPIO_ACTIVE_LOW flag. If a pin was configured as Active Low, physical level low will be considered as logical level 1 (an active state), physical level high will be considered as logical level 0 (an inactive state).

GPIO_INT_DISABLE

Disables GPIO pin interrupt.

GPIO_INT_EDGE_RISING

Configures GPIO interrupt to be triggered on pin rising edge and enables it.

GPIO_INT_EDGE_FALLING

Configures GPIO interrupt to be triggered on pin falling edge and enables it.

GPIO_INT_EDGE_BOTH

Configures GPIO interrupt to be triggered on pin rising or falling edge and enables it.

GPIO_INT_LEVEL_LOW

Configures GPIO interrupt to be triggered on pin physical level low and enables it.

GPIO_INT_LEVEL_HIGH

Configures GPIO interrupt to be triggered on pin physical level high and enables it.

GPIO_INT_EDGE_TO_INACTIVE

Configures GPIO interrupt to be triggered on pin state change to logical level 0 and enables it.

GPIO_INT_EDGE_TO_ACTIVE

Configures GPIO interrupt to be triggered on pin state change to logical level 1 and enables it.

GPIO_INT_LEVEL_INACTIVE

Configures GPIO interrupt to be triggered on pin logical level 0 and enables it.

GPIO_INT_LEVEL_ACTIVE

Configures GPIO interrupt to be triggered on pin logical level 1 and enables it.

GPIO drive strength flags

The GPIO_DS_* flags are used with gpio_pin_configure to specify the drive strength configuration of a GPIO pin.

The drive strength of individual pins can be configured independently for when the pin output is low and high.

The GPIO_DS_*_LOW enumerations define the drive strength of a pin when output is low.

The GPIO_DS_*_HIGH enumerations define the drive strength of a pin when output is high.

The interface supports two different drive strengths: DFLT - The lowest drive strength supported by the HW ALT - The highest drive strength supported by the HW

On hardware that supports only one standard drive strength, both DFLT and ALT have the same behavior.

GPIO_DS_DFLT_LOW

Default drive strength standard when GPIO pin output is low.

GPIO_DS_ALT_LOW

Alternative drive strength when GPIO pin output is low. For hardware that does not support configurable drive strength use the default drive strength.

GPIO_DS_DFLT_HIGH

Default drive strength when GPIO pin output is high.

GPIO_DS_ALT_HIGH

Alternative drive strength when GPIO pin output is high. For hardware that does not support configurable drive strengths use the default drive strength.

GPIO pin active level flags

GPIO_ACTIVE_LOW

GPIO pin is active (has logical value '1') in low state.

GPIO_ACTIVE_HIGH

GPIO pin is active (has logical value '1') in high state.

GPIO pin drive flags

GPIO_OPEN_DRAIN

Configures GPIO output in open drain mode (wired AND).

Note: 'Open Drain' mode also known as 'Open Collector' is an output configuration which behaves like a switch that is either connected to ground or disconnected.

GPIO_OPEN_SOURCE

Configures GPIO output in open source mode (wired OR).

Note: 'Open Source' is a term used by software engineers to describe output mode opposite to 'Open Drain'. It behaves like a switch that is either connected to power supply or disconnected. There exist no corresponding hardware schematic and the term is generally unknown to hardware engineers.

GPIO pin bias flags

GPIO_PULL_UP

Enables GPIO pin pull-up.

GPIO_PULL_DOWN

Enable GPIO pin pull-down.

GPIO pin voltage flags

The voltage flags are a Zephyr specific extension of the standard GPIO flags specified by the Linux GPIO binding. Only applicable if SoC allows to configure pin voltage per individual pin.

GPIO_VOLTAGE_DEFAULT

Set pin at the default voltage level

GPIO_VOLTAGE_1P8

Set pin voltage level at 1.8 V

GPIO_VOLTAGE_3P3

Set pin voltage level at 3.3 V

GPIO_VOLTAGE_5P0

Set pin voltage level at 5.0 V

Defines

GPIO_INT_DEBOUNCE

Enable GPIO pin debounce.

Note: Drivers that do not support a debounce feature should ignore this flag rather than rejecting the configuration with -ENOTSUP.

GPIO_DT_SPEC_GET_BY_IDX(node_id, prop, idx)

Static initializer for a *gpio_dt_spec*.

This returns a static initializer for a *gpio_dt_spec* structure given a devicetree node identifier, a property specifying a GPIO and an index.

Example devicetree fragment:

```
n: node {
   foo-gpios = <&gpio0 1 GPIO_ACTIVE_LOW>,
               <&gpio1 2 GPIO_ACTIVE_LOW>;
}
```

Example usage:

```
const struct gpio_dt_spec spec = GPIO_DT_SPEC_GET_BY_IDX(DT_NODELABEL(n),
                                                         foo_gpios, 1);
// Initializes 'spec' to:
// {
//         .port = DEVICE_DT_GET(DT_NODELABEL(gpio1)),
```

(continues on next page)

(continued from previous page)

```
//          .pin = 2,
//          .dt_flags = GPIO_ACTIVE_LOW
// }
```

The 'gpio' field must still be checked for readiness, e.g. using *device_is_ready()*. It is an error to use this macro unless the node exists, has the given property, and that property specifies a GPIO controller, pin number, and flags as shown above.

Parameters

- `node_id` – devicetree node identifier
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into "prop"

Returns static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_GET_BY_IDX_OR(node_id, prop, idx, default_value)`

Like *GPIO_DT_SPEC_GET_BY_IDX()*, with a fallback to a default value.

If the devicetree node identifier 'node_id' refers to a node with a property 'prop', this expands to *`GPIO_DT_SPEC_GET_BY_IDX(node_id, prop, idx)`*. The `default_value` parameter is not expanded in this case.

Otherwise, this expands to `default_value`.

Parameters

- `node_id` – devicetree node identifier
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into "prop"
- `default_value` – fallback value to expand to

Returns static initializer for a struct *gpio_dt_spec* for the property, or default_value if the node or property do not exist

`GPIO_DT_SPEC_GET(node_id, prop)`

Equivalent to *GPIO_DT_SPEC_GET_BY_IDX(node_id, prop, 0)*.

See also:

GPIO_DT_SPEC_GET_BY_IDX()

Parameters

- `node_id` – devicetree node identifier
- `prop` – lowercase-and-underscores property name

Returns static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_GET_OR(node_id, prop, default_value)`

Equivalent to *GPIO_DT_SPEC_GET_BY_IDX_OR(node_id, prop, 0, default_value)*.

See also:

GPIO_DT_SPEC_GET_BY_IDX_OR()

Parameters

- `node_id` – devicetree node identifier

- `prop` – lowercase-and-underscores property name
- `default_value` – fallback value to expand to

Returns static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_INST_GET_BY_IDX(inst, prop, idx)`

Static initializer for a *gpio_dt_spec* from a DT_DRV_COMPAT instance's GPIO property at an index.

See also:

GPIO_DT_SPEC_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into "prop"

Returns static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_INST_GET_BY_IDX_OR(inst, prop, idx, default_value)`

Static initializer for a *gpio_dt_spec* from a DT_DRV_COMPAT instance's GPIO property at an index, with fallback.

See also:

GPIO_DT_SPEC_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `prop` – lowercase-and-underscores property name
- `idx` – logical index into "prop"
- `default_value` – fallback value to expand to

Returns static initializer for a struct *gpio_dt_spec* for the property

`GPIO_DT_SPEC_INST_GET(inst, prop)`

Equivalent to *GPIO_DT_SPEC_INST_GET_BY_IDX(inst, prop, 0)*.

See also:

GPIO_DT_SPEC_INST_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `prop` – lowercase-and-underscores property name

Returns static initializer for a struct *gpio_dt_spec* for the property

GPIO_DT_SPEC_INST_GET_OR(inst, prop, default_value)

Equivalent to *GPIO_DT_SPEC_INST_GET_BY_IDX_OR(inst, prop, 0, default_value)*.

See also:

GPIO_DT_SPEC_INST_GET_BY_IDX()

Parameters

- `inst` – DT_DRV_COMPAT instance number
- `prop` – lowercase-and-underscores property name
- `default_value` – fallback value to expand to

Returns static initializer for a struct *gpio_dt_spec* for the property

GPIO_MAX_PINS_PER_PORT

Maximum number of pins that are supported by `gpio_port_pins_t`.

Typedefs

typedef uint32_t gpio_port_pins_t

Identifies a set of pins associated with a port.

The pin with index n is present in the set if and only if the bit identified by (1U << n) is set.

typedef uint32_t gpio_port_value_t

Provides values for a set of pins associated with a port.

The value for a pin with index n is high (physical mode) or active (logical mode) if and only if the bit identified by (1U << n) is set. Otherwise the value for the pin is low (physical mode) or inactive (logical mode).

Values of this type are often paired with a `gpio_port_pins_t` value that specifies which encoded pin values are valid for the operation.

typedef uint8_t gpio_pin_t

Provides a type to hold a GPIO pin index.

This reduced-size type is sufficient to record a pin number, e.g. from a devicetree GPIOS property.

typedef uint8_t gpio_dt_flags_t

Provides a type to hold GPIO devicetree flags.

All GPIO flags that can be expressed in devicetree fit in the low 8 bits of the full flags field, so use a reduced-size type to record that part of a GPIOS property.

typedef uint32_t gpio_flags_t

Provides a type to hold GPIO configuration flags.

This type is sufficient to hold all flags used to control GPIO configuration, whether pin or interrupt.

typedef void (*gpio_callback_handler_t)(const struct *device* *port, struct *gpio_callback* *cb, *gpio_port_pins_t* pins)

Define the application callback handler function signature.

Note: cb pointer can be used to retrieve private data through *CONTAINER_OF()* if original struct *gpio_callback* is stored in another private structure.

Param port Device struct for the GPIO device.

Param cb Original struct *gpio_callback* owning this handler

Param pins Mask of pins that triggers the callback handler

Functions

int gpio_pin_interrupt_configure(const struct *device* *port, *gpio_pin_t* pin, *gpio_flags_t* flags)

Configure pin interrupt.

Note: This function can also be used to configure interrupts on pins not controlled directly by the GPIO module. That is, pins which are routed to other modules such as I2C, SPI, UART.

Parameters

- port – Pointer to device structure for the driver instance.
- pin – Pin number.
- flags – Interrupt configuration flags as defined by GPIO_INT_*.

Return values

- 0 – If successful.
- -ENOTSUP – If any of the configuration options is not supported (unless otherwise directed by flag documentation).
- -EINVAL – Invalid argument.
- -EBUSY – Interrupt line required to configure pin interrupt is already in use.
- -EIO – I/O error when accessing an external GPIO chip.
- -EWOULDBLOCK – if operation would block.

static inline int gpio_pin_interrupt_configure_dt(const struct *gpio_dt_spec* *spec, *gpio_flags_t* flags)

Configure pin interrupts from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_interrupt_configure(spec->port, spec->pin, flags);
```

The spec->dt_flags value is not used.

Parameters

- spec – GPIO specification from devicetree
- flags – interrupt configuration flags

Returns a value from *gpio_pin_interrupt_configure()*

int gpio_pin_configure(const struct *device* *port, *gpio_pin_t* pin, *gpio_flags_t* flags)

Configure a single pin.

Parameters

- port – Pointer to device structure for the driver instance.
- pin – Pin number to configure.
- flags – Flags for pin configuration: 'GPIO input/output configuration flags', 'GPIO drive strength flags', 'GPIO pin drive flags', 'GPIO pin bias flags', GPIO_INT_DEBOUNCE.

Return values

- 0 – If successful.
- -ENOTSUP – if any of the configuration options is not supported (unless otherwise directed by flag documentation).
- -EINVAL – Invalid argument.
- -EIO – I/O error when accessing an external GPIO chip.
- -EWOULDBLOCK – if operation would block.

static inline int gpio_pin_configure_dt(const struct *gpio_dt_spec* *spec, *gpio_flags_t* extra_flags)

Configure a single pin from a *gpio_dt_spec* and some extra flags.

This is equivalent to:

```
gpio_pin_configure(spec->port, spec->pin, spec->dt_flags | extra_flags);
```

Parameters

- spec – GPIO specification from devicetree
- extra_flags – additional flags

Returns a value from *gpio_pin_configure()*

int gpio_port_get_raw(const struct *device* *port, *gpio_port_value_t* *value)

Get physical level of all input pins in a port.

A low physical level on the pin will be interpreted as value 0. A high physical level will be interpreted as value 1. This function ignores GPIO_ACTIVE_LOW flag.

Value of a pin with index n will be represented by bit n in the returned port value.

Parameters

- port – Pointer to the device structure for the driver instance.
- value – Pointer to a variable where pin values will be stored.

Return values

- 0 – If successful.
- -EIO – I/O error when accessing an external GPIO chip.
- -EWOULDBLOCK – if operation would block.

static inline int gpio_port_get(const struct *device* *port, *gpio_port_value_t* *value)

Get logical level of all input pins in a port.

Get logical level of an input pin taking into account GPIO_ACTIVE_LOW flag. If pin is configured as Active High, a low physical level will be interpreted as logical value 0. If pin is configured as Active Low, a low physical level will be interpreted as logical value 1.

Value of a pin with index n will be represented by bit n in the returned port value.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `value` – Pointer to a variable where pin values will be stored.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

int `gpio_port_set_masked_raw`(const struct *device* *port, *gpio_port_pins_t* mask, *gpio_port_value_t* value)

Set physical level of output pins in a port.

Writing value 0 to the pin will set it to a low physical level. Writing value 1 will set it to a high physical level. This function ignores GPIO_ACTIVE_LOW flag.

Pin with index n is represented by bit n in mask and value parameter.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `mask` – Mask indicating which pins will be modified.
- `value` – Value assigned to the output pins.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_port_set_masked`(const struct *device* *port, *gpio_port_pins_t* mask, *gpio_port_value_t* value)

Set logical level of output pins in a port.

Set logical level of an output pin taking into account GPIO_ACTIVE_LOW flag. Value 0 sets the pin in logical 0 / inactive state. Value 1 sets the pin in logical 1 / active state. If pin is configured as Active High, the default, setting it in inactive state will force the pin to a low physical level. If pin is configured as Active Low, setting it in inactive state will force the pin to a high physical level.

Pin with index n is represented by bit n in mask and value parameter.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `mask` – Mask indicating which pins will be modified.
- `value` – Value assigned to the output pins.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

int `gpio_port_set_bits_raw`(const struct *device* *port, *gpio_port_pins_t* pins)

Set physical level of selected output pins to high.

Parameters

- `port` – Pointer to the device structure for the driver instance.

- `pins` – Value indicating which pins will be modified.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_port_set_bits`(const struct *device* *port, *gpio_port_pins_t* pins)

Set logical level of selected output pins to active.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pins` – Value indicating which pins will be modified.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

int `gpio_port_clear_bits_raw`(const struct *device* *port, *gpio_port_pins_t* pins)

Set physical level of selected output pins to low.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pins` – Value indicating which pins will be modified.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_port_clear_bits`(const struct *device* *port, *gpio_port_pins_t* pins)

Set logical level of selected output pins to inactive.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pins` – Value indicating which pins will be modified.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

int `gpio_port_toggle_bits`(const struct *device* *port, *gpio_port_pins_t* pins)

Toggle level of selected output pins.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pins` – Value indicating which pins will be modified.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.

- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_port_set_clr_bits_raw`(const struct *device* *port, *gpio_port_pins_t* set_pins, *gpio_port_pins_t* clear_pins)

Set physical level of selected output pins.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `set_pins` – Value indicating which pins will be set to high.
- `clear_pins` – Value indicating which pins will be set to low.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_port_set_clr_bits`(const struct *device* *port, *gpio_port_pins_t* set_pins, *gpio_port_pins_t* clear_pins)

Set logical level of selected output pins.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `set_pins` – Value indicating which pins will be set to active.
- `clear_pins` – Value indicating which pins will be set to inactive.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_pin_get_raw`(const struct *device* *port, *gpio_pin_t* pin)

Get physical level of an input pin.

A low physical level on the pin will be interpreted as value 0. A high physical level will be interpreted as value 1. This function ignores GPIO_ACTIVE_LOW flag.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pin` – Pin number.

Return values

- `1` – If pin physical level is high.
- `0` – If pin physical level is low.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_pin_get`(const struct *device* *port, *gpio_pin_t* pin)

Get logical level of an input pin.

Get logical level of an input pin taking into account GPIO_ACTIVE_LOW flag. If pin is configured as Active High, a low physical level will be interpreted as logical value 0. If pin is configured as Active Low, a low physical level will be interpreted as logical value 1.

Note: If pin is configured as Active High, the default, *gpio_pin_get()* function is equivalent to *gpio_pin_get_raw()*.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pin` – Pin number.

Return values

- `1` – If pin logical value is 1 / active.
- `0` – If pin logical value is 0 / inactive.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_pin_get_dt`(const struct *gpio_dt_spec* *spec)

Get logical level of an input pin from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_get(spec->port, spec->pin);
```

Parameters

- `spec` – GPIO specification from devicetree

Returns a value from *gpio_pin_get()*

static inline int `gpio_pin_set_raw`(const struct *device* *port, *gpio_pin_t* pin, int value)

Set physical level of an output pin.

Writing value 0 to the pin will set it to a low physical level. Writing any value other than 0 will set it to a high physical level. This function ignores GPIO_ACTIVE_LOW flag.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pin` – Pin number.
- `value` – Value assigned to the pin.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_pin_set`(const struct *device* *port, *gpio_pin_t* pin, int value)

Set logical level of an output pin.

Set logical level of an output pin taking into account GPIO_ACTIVE_LOW flag. Value 0 sets the pin in logical 0 / inactive state. Any value other than 0 sets the pin in logical 1 / active state. If pin is configured as Active High, the default, setting it in inactive state will force the pin to a low physical level. If pin is configured as Active Low, setting it in inactive state will force the pin to a high physical level.

Note: If pin is configured as Active High, *gpio_pin_set()* function is equivalent to *gpio_pin_set_raw()*.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pin` – Pin number.
- `value` – Value assigned to the pin.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_pin_set_dt`(const struct *gpio_dt_spec* *spec, int value)

Set logical level of a output pin from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_set(spec->port, spec->pin, value);
```

Parameters

- `spec` – GPIO specification from devicetree
- `value` – Value assigned to the pin.

Returns a value from *gpio_pin_set()*

static inline int `gpio_pin_toggle`(const struct *device* *port, *gpio_pin_t* pin)

Toggle pin level.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `pin` – Pin number.

Return values

- `0` – If successful.
- `-EIO` – I/O error when accessing an external GPIO chip.
- `-EWOULDBLOCK` – if operation would block.

static inline int `gpio_pin_toggle_dt`(const struct *gpio_dt_spec* *spec)

Toggle pin level from a *gpio_dt_spec*.

This is equivalent to:

```
gpio_pin_toggle(spec->port, spec->pin);
```

Parameters

- `spec` – GPIO specification from devicetree

Returns a value from *gpio_pin_toggle()*

static inline void `gpio_init_callback`(struct *gpio_callback* *callback, *gpio_callback_handler_t* handler, *gpio_port_pins_t* pin_mask)

Helper to initialize a struct *gpio_callback* properly.

Parameters

- `callback` – A valid Application's callback structure pointer.
- `handler` – A valid handler function pointer.
- `pin_mask` – A bit mask of relevant pins for the handler

static inline int gpio_add_callback(const struct *device* *port, struct *gpio_callback* *callback)

Add an application callback.

Note: enables to add as many callback as needed on the same port.

Note: Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current GPIO event is not specified.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `callback` – A valid Application's callback structure pointer.

Returns 0 if successful, negative errno code on failure.

static inline int gpio_remove_callback(const struct *device* *port, struct *gpio_callback* *callback)

Remove an application callback.

Note: enables to remove as many callbacks as added through *gpio_add_callback()*.

Warning: It is explicitly permitted, within a callback handler, to remove the registration for the callback that is running, i.e. `callback`. Attempts to remove other registrations on the same device may result in undefined behavior, including failure to invoke callbacks that remain registered and unintended invocation of removed callbacks.

Parameters

- `port` – Pointer to the device structure for the driver instance.
- `callback` – A valid application's callback structure pointer.

Returns 0 if successful, negative errno code on failure.

int gpio_get_pending_int(const struct *device* *dev)

Function to get pending interrupts.

The purpose of this function is to return the interrupt status register for the device. This is especially useful when waking up from low power states to check the wake up source.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `status` – != 0 if at least one gpio interrupt is pending.
- `0` – if no gpio interrupt is pending.

struct gpio_dt_spec

#include <gpio.h> Provides a type to hold GPIO information specified in devicetree.

This type is sufficient to hold a GPIO device pointer, pin number, and the subset of the flags used to control GPIO configuration which may be given in devicetree.

struct gpio_driver_config

#include <gpio.h> This structure is common to all GPIO drivers and is expected to be the first element in the object pointed to by the config field in the device structure.

struct gpio_driver_data

#include <gpio.h> This structure is common to all GPIO drivers and is expected to be the first element in the driver's struct driver_data declaration.

struct gpio_callback

#include <gpio.h> GPIO callback structure.

Used to register a callback in the driver instance callback list. As many callbacks as needed can be added as long as each of them are unique pointers of struct *gpio_callback*. Beware such structure should not be allocated on stack.

Note: To help setting it, see *gpio_init_callback()* below

Public Members

sys_snode_t node

This is meant to be used in the driver and the user should not mess with it (see drivers/gpio/gpio_utils.h)

gpio_callback_handler_t handler

Actual callback function being called when relevant.

gpio_port_pins_t pin_mask

A mask of pins the callback is interested in, if 0 the callback will never be called. Such pin_mask can be modified whenever necessary by the owner, and thus will affect the handler being called or not. The selected pins must be configured to trigger an interrupt.

7.21.12 Hardware Information

Overview

The HW Info API provides access to hardware information such as device identifiers and reset cause flags.

Reset cause flags can be used to determine why the device was reset; for example due to a watchdog timeout or due to power cycling. Different devices support different subset of flags. Use *hwinfo_get_supported_reset_cause* to retrieve the flags that are supported by that device.

Configuration Options

Related configuration options:

- CONFIG_HWINFO

API Reference

group hwinfo_interface

Hardware Information Interface.

Defines

RESET_PIN

RESET_SOFTWARE

RESET_BROWNOUT

RESET_POR

RESET_WATCHDOG

RESET_DEBUG

RESET_SECURITY

RESET_LOW_POWER_WAKE

RESET_CPU_LOCKUP

RESET_PARITY

RESET_PLL

RESET_CLOCK

Functions

ssize_t `hwinfo_get_device_id`(uint8_t *buffer, size_t length)

Copy the device id to a buffer.

This routine copies "length" number of bytes of the device ID to the buffer. If the device ID is smaller then length, the rest of the buffer is left unchanged. The ID depends on the hardware and is not guaranteed unique.

Drivers are responsible for ensuring that the ID data structure is a sequence of bytes. The returned ID value is not supposed to be interpreted based on vendor-specific assumptions of byte order. It should express the identifier as a raw byte sequence, doing any endian conversion necessary so that a hex representation of the bytes produces the intended serial number.

Parameters

- `buffer` – Buffer to write the ID to.
- `length` – Max length of the buffer.

Return values

- `size` – of the device ID copied.
- `-ENOTSUP` – if there is no implementation for the particular device.
- `any` – negative value on driver specific errors.

int `hwinfo_get_reset_cause`(uint32_t *cause)

Retrieve cause of device reset.

This routine retrieves the flags that indicate why the device was reset.

On some platforms the reset cause flags accumulate between successive resets and this routine may return multiple flags indicating all reset causes since the device was powered on. If you need to retrieve the cause only for the most recent reset call `hwinfo_clear_reset_cause` after calling this routine to clear the hardware flags before the next reset event.

Successive calls to this routine will return the same value, unless `hwinfo_clear_reset_cause` has been called.

Parameters

- `cause` – OR'd `reset_cause` flags

Return values

- `zero` – if successful.
- `-ENOTSUP` – if there is no implementation for the particular device.
- `any` – negative value on driver specific errors.

int `hwinfo_clear_reset_cause`(void)

Clear cause of device reset.

Clears reset cause flags.

Return values

- `zero` – if successful.
- `-ENOTSUP` – if there is no implementation for the particular device.
- `any` – negative value on driver specific errors.

int `hwinfo_get_supported_reset_cause`(uint32_t *supported)

Get supported reset cause flags.

Retrieves all `reset_cause` flags that are supported by this device.

Parameters

- `supported` – OR'd `reset_cause` flags that are supported

Return values

- `zero` – if successful.
- `-ENOTSUP` – if there is no implementation for the particular device.
- `any` – negative value on driver specific errors.

7.21.13 I2C EEPROM Slave

Overview

API Reference

group `i2c_eeprom_slave_api`

I2C EEPROM Slave Driver API.

Functions

int `eeprom_slave_program`(const struct *device* *dev, const uint8_t *eeprom_data, unsigned int length)

Program memory of the virtual EEPROM.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `eeprom_data` – Pointer of data to program into the virtual eeprom memory
- `length` – Length of data to program into the virtual eeprom memory

Return values

- `0` – If successful.
- `-EINVAL` – Invalid data size

int `eeprom_slave_read`(const struct *device* *dev, uint8_t *eeprom_data, unsigned int offset)

Read single byte of virtual EEPROM memory.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `eeprom_data` – Pointer of byte where to store the virtual eeprom memory
- `offset` – Offset into EEPROM memory where to read the byte

Return values

- `0` – If successful.
- `-EINVAL` – Invalid data pointer or offset

7.21.14 I2C

Overview

Note: Zephyr recognizes the need to change the terms “master” and “slave” used in the current I2C Specification. This will be done when the conditions identified in *Rule A.2: Inclusive Language* have been met. Existing documentation, data structures, functions, and value symbols in code are likely to change at that point.

I2C (Inter-Integrated Circuit, pronounced “eye squared see”) is a commonly-used two-signal shared peripheral interface bus. Many system-on-chip solutions provide controllers that communicate on an I2C bus. Devices on the bus can operate in two roles: as a “master” that initiates transactions and controls the clock, or as a “slave” that responds to transaction commands. A I2C controller on a given SoC will generally support the master role, and some will also support the slave mode. Zephyr has API for both roles.

I2C Master API Zephyr’s I2C master API is used when an I2C peripheral controls the bus, in particularly the start and stop conditions and the clock. This is the most common mode, used to interact with I2C devices like sensors and serial memory.

This API is supported in all in-tree I2C peripheral drivers and is considered stable.

I2C Slave API Zephyr's I2C slave API is used when an I2C peripheral responds to transactions initiated by a different controller on the bus. It might be used for a Zephyr application with transducer roles that are controlled by another device such as a host processor.

This API is supported in very few in-tree I2C peripheral drivers. The API is considered experimental, as it is not compatible with the capabilities of all I2C peripherals supported in master mode.

Configuration Options

Related configuration options:

- `CONFIG_I2C`

API Reference

group `i2c_interface`

I2C Interface.

Defines

`I2C_SPEED_STANDARD`

I2C Standard Speed: 100 kHz

`I2C_SPEED_FAST`

I2C Fast Speed: 400 kHz

`I2C_SPEED_FAST_PLUS`

I2C Fast Plus Speed: 1 MHz

`I2C_SPEED_HIGH`

I2C High Speed: 3.4 MHz

`I2C_SPEED_ULTRA`

I2C Ultra Fast Speed: 5 MHz

`I2C_SPEED_SHIFT`

`I2C_SPEED_SET(speed)`

`I2C_SPEED_MASK`

`I2C_SPEED_GET(cfg)`

`I2C_ADDR_10_BITS`

Use 10-bit addressing. DEPRECATED - Use I2C_MSG_ADDR_10_BITS instead.

`I2C_MODE_MASTER`

Controller to act as Master.

I2C_DT_SPEC_GET(node_id)

Structure initializer for *i2c_dt_spec* from devicetree.

This helper macro expands to a static initializer for a `struct i2c_dt_spec` by reading the relevant bus and address data from the devicetree.

Parameters

- `node_id` – Devicetree node identifier for the I2C device whose struct *i2c_dt_spec* to create an initializer for

I2C_DT_SPEC_INST_GET(inst)

Structure initializer for *i2c_dt_spec* from devicetree instance.

This is equivalent to *I2C_DT_SPEC_GET(DT_DRV_INST(inst))*.

Parameters

- `inst` – Devicetree instance number

I2C_MSG_WRITE

Write message to I2C bus.

I2C_MSG_READ

Read message from I2C bus.

I2C_MSG_STOP

Send STOP after this message.

I2C_MSG_RESTART

RESTART I2C transaction for this message.

Note: Not all I2C drivers have or require explicit support for this feature. Some drivers require this be present on a read message that follows a write, or vice-versa. Some drivers will merge adjacent fragments into a single transaction using this flag; some will not.

I2C_MSG_ADDR_10_BITS

Use 10-bit addressing for this message.

Note: Not all SoC I2C implementations support this feature.

I2C_SLAVE_FLAGS_ADDR_10_BITS

Slave device responds to 10-bit addressing.

I2C_DECLARE_CLIENT_CONFIG

I2C_CLIENT(_master, _addr)

I2C_GET_MASTER(_conf)

I2C_GET_ADDR(_conf)

Typedefs

typedef int (*i2c_slave_write_requested_cb_t)(struct *i2c_slave_config* *config)

Function called when a write to the device is initiated.

This function is invoked by the controller when the bus completes a start condition for a write operation to the address associated with a particular device.

A success return shall cause the controller to ACK the next byte received. An error return shall cause the controller to NACK the next byte received.

Param config the configuration structure associated with the device to which the operation is addressed.

Return 0 if the write is accepted, or a negative error code.

typedef int (*i2c_slave_write_received_cb_t)(struct *i2c_slave_config* *config, uint8_t val)

Function called when a write to the device is continued.

This function is invoked by the controller when it completes reception of a byte of data in an ongoing write operation to the device.

A success return shall cause the controller to ACK the next byte received. An error return shall cause the controller to NACK the next byte received.

Param config the configuration structure associated with the device to which the operation is addressed.

Param val the byte received by the controller.

Return 0 if more data can be accepted, or a negative error code.

typedef int (*i2c_slave_read_requested_cb_t)(struct *i2c_slave_config* *config, uint8_t *val)

Function called when a read from the device is initiated.

This function is invoked by the controller when the bus completes a start condition for a read operation from the address associated with a particular device.

The value returned in `*val` will be transmitted. A success return shall cause the controller to react to additional read operations. An error return shall cause the controller to ignore bus operations until a new start condition is received.

Param config the configuration structure associated with the device to which the operation is addressed.

Param val pointer to storage for the first byte of data to return for the read request.

Return 0 if more data can be requested, or a negative error code.

typedef int (*i2c_slave_read_processed_cb_t)(struct *i2c_slave_config* *config, uint8_t *val)

Function called when a read from the device is continued.

This function is invoked by the controller when the bus is ready to provide additional data for a read operation from the address associated with the device device.

The value returned in `*val` will be transmitted. A success return shall cause the controller to react to additional read operations. An error return shall cause the controller to ignore bus operations until a new start condition is received.

Param config the configuration structure associated with the device to which the operation is addressed.

Param val pointer to storage for the next byte of data to return for the read request.

Return 0 if data has been provided, or a negative error code.

typedef int (*i2c_slave_stop_cb_t)(struct *i2c_slave_config* *config)

Function called when a stop condition is observed after a start condition addressed to a particular device.

This function is invoked by the controller when the bus is ready to provide additional data for a read operation from the address associated with the device device. After the function returns the controller shall enter a state where it is ready to react to new start conditions.

Param config the configuration structure associated with the device to which the operation is addressed.

Return Ignored.

Functions

int i2c_configure(const struct *device* *dev, uint32_t dev_config)

Configure operation of a host controller.

Parameters

- dev – Pointer to the device structure for the driver instance.
- dev_config – Bit-packed 32-bit value to the device runtime configuration for the I2C controller.

Return values

- 0 – If successful.
- -EIO – General input / output error, failed to configure device.

int i2c_transfer(const struct *device* *dev, struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr)

Perform data transfer to another I2C device in master mode.

This routine provides a generic interface to perform data transfer to another I2C device synchronously. Use *i2c_read()*/i2c_write() for simple read or write.

The array of message *msgs* must not be NULL. The number of message *num_msgs* may be zero,in which case no transfer occurs.

Note: Not all scatter/gather transactions can be supported by all drivers. As an example, a gather write (multiple consecutive *i2c_msg* buffers all configured for I2C_MSG_WRITE) may be packed into a single transaction by some drivers, but others may emit each fragment as a distinct write transaction, which will not produce the same behavior. See the documentation of struct *i2c_msg* for limitations on support for multi-message bus transactions.

Parameters

- dev – Pointer to the device structure for an I2C controller driver configured in master mode.
- msgs – Array of messages to transfer.
- num_msgs – Number of messages to transfer.
- addr – Address of the I2C target device.

Return values

- 0 – If successful.
- -EIO – General input / output error.

static inline int i2c_transfer_dt(const struct *i2c_dt_spec* *spec, struct *i2c_msg* *msgs, uint8_t num_msgs)

Perform data transfer to another I2C device in master mode.

This is equivalent to:

```
i2c_transfer(spec->bus, msgs, num_msgs, spec->addr);
```

Parameters

- spec – I2C specification from devicetree.
- msgs – Array of messages to transfer.
- num_msgs – Number of messages to transfer.

Returns a value from *i2c_transfer()*

int i2c_recover_bus(const struct *device* *dev)

Recover the I2C bus.

Attempt to recover the I2C bus.

Parameters

- dev – Pointer to the device structure for an I2C controller driver configured in master mode.

Return values

- 0 – If successful
- -EBUSY – If bus is not clear after recovery attempt.
- -EIO – General input / output error.
- -ENOSYS – If bus recovery is not implemented

static inline int i2c_slave_register(const struct *device* *dev, struct *i2c_slave_config* *cfg)

Registers the provided config as Slave device of a controller.

Enable I2C slave mode for the 'dev' I2C bus driver using the provided 'config' struct containing the functions and parameters to send bus events. The I2C slave will be registered at the address provided as 'address' struct member. Addressing mode - 7 or 10 bit - depends on the 'flags' struct member. Any I2C bus events related to the slave mode will be passed onto I2C slave device driver via a set of callback functions provided in the 'callbacks' struct member.

Most of the existing hardware allows simultaneous support for master and slave mode. This is however not guaranteed.

Parameters

- dev – Pointer to the device structure for an I2C controller driver configured in slave mode.
- cfg – Config struct with functions and parameters used by the I2C driver to send bus events

Return values

- 0 – Is successful
- -EINVAL – If parameters are invalid
- -EIO – General input / output error.
- -ENOSYS – If slave mode is not implemented

static inline int `i2c_slave_unregister`(const struct *device* *dev, struct *i2c_slave_config* *cfg)

Unregisters the provided config as Slave device.

This routine disables I2C slave mode for the 'dev' I2C bus driver using the provided 'config' struct containing the functions and parameters to send bus events.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in slave mode.
- `cfg` – Config struct with functions and parameters used by the I2C driver to send bus events

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid
- `-ENOSYS` – If slave mode is not implemented

int `i2c_slave_driver_register`(const struct *device* *dev)

Instructs the I2C Slave device to register itself to the I2C Controller.

This routine instructs the I2C Slave device to register itself to the I2C Controller via its parent controller's *i2c_slave_register()* API.

Parameters

- `dev` – Pointer to the device structure for the I2C slave device (not itself an I2C controller).

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid
- `-EIO` – General input / output error.

int `i2c_slave_driver_unregister`(const struct *device* *dev)

Instructs the I2C Slave device to unregister itself from the I2C Controller.

This routine instructs the I2C Slave device to unregister itself from the I2C Controller via its parent controller's *i2c_slave_register()* API.

Parameters

- `dev` – Pointer to the device structure for the I2C slave device (not itself an I2C controller).

Return values

- `0` – Is successful
- `-EINVAL` – If parameters are invalid

static inline int `i2c_write`(const struct *device* *dev, const uint8_t *buf, uint32_t num_bytes, uint16_t addr)

Write a set amount of data to an I2C device.

This routine writes a set amount of data synchronously.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `buf` – Memory pool from which the data is transferred.
- `num_bytes` – Number of bytes to write.

- `addr` – Address to the target I2C device for writing.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_write_dt`(const struct *i2c_dt_spec* *spec, const uint8_t *buf, uint32_t num_bytes)

Write a set amount of data to an I2C device.

This is equivalent to:

```
i2c_write(spec->bus, buf, num_bytes, spec->addr);
```

Parameters

- `spec` – I2C specification from devicetree.
- `buf` – Memory pool from which the data is transferred.
- `num_bytes` – Number of bytes to write.

Returns a value from *i2c_write()*

static inline int `i2c_read`(const struct *device* *dev, uint8_t *buf, uint32_t num_bytes, uint16_t addr)

Read a set amount of data from an I2C device.

This routine reads a set amount of data synchronously.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `buf` – Memory pool that stores the retrieved data.
- `num_bytes` – Number of bytes to read.
- `addr` – Address of the I2C device being read.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_read_dt`(const struct *i2c_dt_spec* *spec, uint8_t *buf, uint32_t num_bytes)

Read a set amount of data from an I2C device.

This is equivalent to:

```
i2c_read(spec->bus, buf, num_bytes, spec->addr);
```

Parameters

- `spec` – I2C specification from devicetree.
- `buf` – Memory pool that stores the retrieved data.
- `num_bytes` – Number of bytes to read.

Returns a value from *i2c_read()*

static inline int `i2c_write_read`(const struct *device* *dev, uint16_t addr, const void *write_buf, size_t num_write, void *read_buf, size_t num_read)

Write then read data from an I2C device.

This supports the common operation "this is what I want", "now give

it to me" transaction pair through a combined write-then-read bus transaction.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `addr` – Address of the I2C device
- `write_buf` – Pointer to the data to be written
- `num_write` – Number of bytes to write
- `read_buf` – Pointer to storage for read data
- `num_read` – Number of bytes to read

Return values

- `0` – if successful
- `negative` – on error.

static inline int `i2c_write_read_dt`(const struct *i2c_dt_spec* *spec, const void *write_buf, size_t num_write, void *read_buf, size_t num_read)

Write then read data from an I2C device.

This is equivalent to:

```
i2c_write_read(spec->bus, spec->addr,
               write_buf, num_write,
               read_buf, num_read);
```

Parameters

- `spec` – I2C specification from devicetree.
- `write_buf` – Pointer to the data to be written
- `num_write` – Number of bytes to write
- `read_buf` – Pointer to storage for read data
- `num_read` – Number of bytes to read

Returns a value from *i2c_write_read()*

static inline int `i2c_burst_read`(const struct *device* *dev, uint16_t dev_addr, uint8_t start_addr, uint8_t *buf, uint32_t num_bytes)

Read multiple bytes from an internal address of an I2C device.

This routine reads multiple bytes from an internal address of an I2C device synchronously.

Instances of this may be replaced by *i2c_write_read()*.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `dev_addr` – Address of the I2C device for reading.

- `start_addr` – Internal address from which the data is being read.
- `buf` – Memory pool that stores the retrieved data.
- `num_bytes` – Number of bytes being read.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_burst_read_dt`(const struct *i2c_dt_spec* *spec, uint8_t start_addr, uint8_t *buf, uint32_t num_bytes)

Read multiple bytes from an internal address of an I2C device.

This is equivalent to:

```
i2c_burst_read(spec->bus, spec->addr, start_addr, buf, num_bytes);
```

Parameters

- `spec` – I2C specification from devicetree.
- `start_addr` – Internal address from which the data is being read.
- `buf` – Memory pool that stores the retrieved data.
- `num_bytes` – Number of bytes to read.

Returns a value from *i2c_burst_read()*

static inline int `i2c_burst_write`(const struct *device* *dev, uint16_t dev_addr, uint8_t start_addr, const uint8_t *buf, uint32_t num_bytes)

Write multiple bytes to an internal address of an I2C device.

This routine writes multiple bytes to an internal address of an I2C device synchronously.

Warning: The combined write synthesized by this API may not be supported on all I2C devices. Uses of this API may be made more portable by replacing them with calls to *i2c_write()* passing a buffer containing the combined address and data.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `dev_addr` – Address of the I2C device for writing.
- `start_addr` – Internal address to which the data is being written.
- `buf` – Memory pool from which the data is transferred.
- `num_bytes` – Number of bytes being written.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_burst_write_dt`(const struct *i2c_dt_spec* *spec, uint8_t start_addr, const uint8_t *buf, uint32_t num_bytes)

Write multiple bytes to an internal address of an I2C device.

This is equivalent to:

```
i2c_burst_write(spec->bus, spec->addr, start_addr, buf, num_bytes);
```

Parameters

- `spec` – I2C specification from devicetree.
- `start_addr` – Internal address to which the data is being written.
- `buf` – Memory pool from which the data is transferred.
- `num_bytes` – Number of bytes being written.

Returns a value from *i2c_burst_write()*

static inline int `i2c_reg_read_byte`(const struct *device* *dev, uint16_t dev_addr, uint8_t reg_addr, uint8_t *value)

Read internal register of an I2C device.

This routine reads the value of an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `dev_addr` – Address of the I2C device for reading.
- `reg_addr` – Address of the internal register being read.
- `value` – Memory pool that stores the retrieved register value.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_reg_read_byte_dt`(const struct *i2c_dt_spec* *spec, uint8_t reg_addr, uint8_t *value)

Read internal register of an I2C device.

This is equivalent to:

```
i2c_reg_read_byte(spec->bus, spec->addr, reg_addr, value);
```

Parameters

- `spec` – I2C specification from devicetree.
- `reg_addr` – Address of the internal register being read.
- `value` – Memory pool that stores the retrieved register value.

Returns a value from *i2c_reg_read_byte()*

static inline int `i2c_reg_write_byte`(const struct *device* *dev, uint16_t dev_addr, uint8_t reg_addr, uint8_t value)

Write internal register of an I2C device.

This routine writes a value to an 8-bit internal register of an I2C device synchronously.

Note: This function internally combines the register and value into a single bus transaction.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `dev_addr` – Address of the I2C device for writing.
- `reg_addr` – Address of the internal register being written.
- `value` – Value to be written to internal register.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_reg_write_byte_dt`(const struct *i2c_dt_spec* *spec, uint8_t reg_addr, uint8_t value)

Write internal register of an I2C device.

This is equivalent to:

```
i2c_reg_write_byte(spec->bus, spec->addr, reg_addr, value);
```

Parameters

- `spec` – I2C specification from devicetree.
- `reg_addr` – Address of the internal register being written.
- `value` – Value to be written to internal register.

Returns a value from *i2c_reg_write_byte()*

static inline int `i2c_reg_update_byte`(const struct *device* *dev, uint8_t dev_addr, uint8_t reg_addr, uint8_t mask, uint8_t value)

Update internal register of an I2C device.

This routine updates the value of a set of bits from an 8-bit internal register of an I2C device synchronously.

Note: If the calculated new register value matches the value that was read this function will not generate a write operation.

Parameters

- `dev` – Pointer to the device structure for an I2C controller driver configured in master mode.
- `dev_addr` – Address of the I2C device for updating.
- `reg_addr` – Address of the internal register being updated.
- `mask` – Bitmask for updating internal register.
- `value` – Value for updating internal register.

Return values

- `0` – If successful.
- `-EIO` – General input / output error.

static inline int `i2c_reg_update_byte_dt`(const struct *i2c_dt_spec* *spec, uint8_t reg_addr, uint8_t mask, uint8_t value)

Update internal register of an I2C device.

This is equivalent to:

```
i2c_reg_update_byte(spec->bus, spec->addr, reg_addr, mask, value);
```

Parameters

- `spec` – I2C specification from devicetree.
- `reg_addr` – Address of the internal register being updated.
- `mask` – Bitmask for updating internal register.
- `value` – Value for updating internal register.

Returns a value from *i2c_reg_update_byte()*

void `i2c_dump_msgs`(const char *name, const struct *i2c_msg* *msgs, uint8_t num_msgs, uint16_t addr)

Dump out an I2C message.

Dumps out a list of I2C messages. For any that are writes (W), the data is displayed in hex.

It looks something like this (with name "testing"):

D: I2C msg: testing, addr=56 D: W len=01: D: contents: D: 06 |. D: W len=0e: D: contents: D: 00 01 02 03 04 05 06 07 |........ D: 08 09 0a 0b 0c 0d |......

Parameters

- `name` – Name of this dump, displayed at the top.
- `msgs` – Array of messages to dump.
- `num_msgs` – Number of messages to dump.
- `addr` – Address of the I2C target device.

struct `i2c_dt_spec`

#include <i2c.h> Complete I2C DT information.

Param bus is the I2C bus

Param addr is the slave address

struct `i2c_msg`

#include <i2c.h> One I2C Message.

This defines one I2C message to transact on the I2C bus.

Note: Some of the configurations supported by this API may not be supported by specific SoC I2C hardware implementations, in particular features related to bus transactions intended to read or write data from different buffers within a single transaction. Invocations of *i2c_transfer()* may not indicate an error when an unsupported configuration is encountered. In some cases drivers will generate separate transactions for each message fragment, with or without presence of *I2C_MSG_RESTART* in *flags*.

Public Members

uint8_t *buf

Data buffer in bytes

uint32_t len

Length of buffer in bytes

uint8_t flags

Flags for this message

struct i2c_slave_callbacks

#include <i2c.h> Structure providing callbacks to be implemented for devices that supports the I2C slave API.

This structure may be shared by multiple devices that implement the same API at different addresses on the bus.

struct i2c_slave_config

#include <i2c.h> Structure describing a device that supports the I2C slave API.

Instances of this are passed to the *i2c_slave_register()* and *i2c_slave_unregister()* functions to indicate addition and removal of a slave device, respective.

Fields other than node must be initialized by the module that implements the device behavior prior to passing the object reference to *i2c_slave_register()*.

Public Members

sys_snode_t node

Private, do not modify

uint8_t flags

Flags for the slave device defined by I2C_SLAVE_FLAGS_* constants

uint16_t address

Address for this slave device

const struct *i2c_slave_callbacks* *callbacks

Callback functions

struct i2c_client_config

#include <i2c.h>

7.21.15 IPM

Overview

API Reference

group `ipm_interface`

IPM Interface.

Typedefs

typedef void (*`ipm_callback_t`)(const struct *device* *ipmdev, void *user_data, uint32_t id, volatile void *data)

Callback API for incoming IPM messages.

These callbacks execute in interrupt context. Therefore, use only interrupt-safe APIS. Registration of callbacks is done via *ipm_register_callback*

Param ipmdev Driver instance

Param user_data Pointer to some private data provided at registration time.

Param id Message type identifier.

Param data Message data pointer. The correct amount of data to read out must be inferred using the message id/upper level protocol.

typedef int (*`ipm_send_t`)(const struct *device* *ipmdev, int wait, uint32_t id, const void *data, int size)

Callback API to send IPM messages.

See ipm_send() for argument definitions.

typedef int (*`ipm_max_data_size_get_t`)(const struct *device* *ipmdev)

Callback API to get maximum data size.

See ipm_max_data_size_get() for argument definitions.

typedef uint32_t (*`ipm_max_id_val_get_t`)(const struct *device* *ipmdev)

Callback API to get the ID's maximum value.

See ipm_max_id_val_get() for argument definitions.

typedef void (*`ipm_register_callback_t`)(const struct *device* *port, *ipm_callback_t* cb, void *user_data)

Callback API upon registration.

See ipm_register_callback() for argument definitions.

typedef int (*`ipm_set_enabled_t`)(const struct *device* *ipmdev, int enable)

Callback API upon enablement of interrupts.

See ipm_set_enabled() for argument definitions.

Functions

int `ipm_send`(const struct *device* *ipmdev, int wait, uint32_t id, const void *data, int size)

Try to send a message over the IPM device.

A message is considered consumed once the remote interrupt handler finishes. If there is deferred processing on the remote side, or if outgoing messages must be queued and wait on an event/semaphore, a high-level driver can implement that.

There are constraints on how much data can be sent or the maximum value of id. Use the *ipm_max_data_size_get* and *ipm_max_id_val_get* routines to determine them.

The *size* parameter is used only on the sending side to determine the amount of data to put in the message registers. It is not passed along to the receiving side. The upper-level protocol dictates the amount of data read back.

Parameters

- `ipmdev` – Driver instance
- `wait` – If nonzero, busy-wait for remote to consume the message. The message is considered consumed once the remote interrupt handler finishes. If there is deferred processing on the remote side, or you would like to queue outgoing messages and wait on an event/semaphore, you can implement that in a high-level driver
- `id` – Message identifier. Values are constrained by *ipm_max_data_size_get* since many boards only allow for a subset of bits in a 32-bit register to store the ID.
- `data` – Pointer to the data sent in the message.
- `size` – Size of the data.

Return values

- `-EBUSY` – If the remote hasn't yet read the last data sent.
- `-EMSGSIZE` – If the supplied data size is unsupported by the driver.
- `-EINVAL` – If there was a bad parameter, such as: too-large id value. or the device isn't an outbound IPM channel.
- `0` – On success.

static inline void `ipm_register_callback`(const struct *device* *ipmdev, *ipm_callback_t* cb, void *user_data)

Register a callback function for incoming messages.

Parameters

- `ipmdev` – Driver instance pointer.
- `cb` – Callback function to execute on incoming message interrupts.
- `user_data` – Application-specific data pointer which will be passed to the callback function when executed.

int `ipm_max_data_size_get`(const struct *device* *ipmdev)

Return the maximum number of bytes possible in an outbound message.

IPM implementations vary on the amount of data that can be sent in a single message since the data payload is typically stored in registers.

Parameters

- `ipmdev` – Driver instance pointer.

Returns Maximum possible size of a message in bytes.

uint32_t `ipm_max_id_val_get`(const struct *device* *ipmdev)

Return the maximum id value possible in an outbound message.

Many IPM implementations store the message's ID in a register with some bits reserved for other uses.

Parameters

- `ipmdev` – Driver instance pointer.

Returns Maximum possible value of a message ID.

int ipm_set_enabled(const struct *device* *ipmdev, int enable)

Enable interrupts and callbacks for inbound channels.

Parameters

- ipmdev – Driver instance pointer.
- enable – Set to 0 to disable and to nonzero to enable.

Return values

- 0 – On success.
- -EINVAL – If it isn't an inbound channel.

struct ipm_driver_api

#include <ipm.h>

7.21.16 KSCAN

Overview

The kscan driver (keyboard scan matrix) is used for detecting a key press in a connected matrix keyboard or any device with buttons such as joysticks. Typically, matrix keyboards are implemented using a two-dimensional configuration in order to sense several keys. This allows interfacing to many keys through fewer physical pins. Keyboard matrix drivers read the rows while applying power through the columns one at a time with the purpose of detecting key events. There is no correlation between the physical and electrical layout of keys. For, example, the physical layout may be one array of 16 or fewer keys, which may be electrically connected to a 4 x 4 array. In addition, key values are defined by a keymap provided by the keyboard manufacturer.

Configuration Options

Related configuration options:

- CONFIG_KSCAN

API Reference

group kscan_interface

KSCAN APIs.

Typedefs

typedef void (*kscan_callback_t)(const struct *device* *dev, uint32_t row, uint32_t column, bool pressed)

Keyboard scan callback called when user press/release a key on a matrix keyboard.

Param dev Pointer to the device structure for the driver instance.

Param row Describes row change.

Param column Describes column change.

Param pressed Describes the kind of key event.

Functions

int `kscan_config`(const struct *device* *dev, *kscan_callback_t* callback)

Configure a Keyboard scan instance.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – called when keyboard devices reply to to a keyboard event such as key pressed/released.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `kscan_enable_callback`(const struct *device* *dev)

Enables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `kscan_disable_callback`(const struct *device* *dev)

Disables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

7.21.17 LED

Overview

The LED API provides access to Light Emitting Diodes, both in individual and stip form.

Configuration Options

Related configuration options:

- `CONFIG_LED`
- `CONFIG_LED_STRIP`

API Reference

LED

group `led_interface`

LED Interface.

Typedefs

typedef int (*`led_api_blink`)(const struct *device* *dev, uint32_t led, uint32_t delay_on, uint32_t delay_off)

Callback API for blinking an LED.

See also:

led_blink() for argument descriptions.

typedef int (*`led_api_get_info`)(const struct *device* *dev, uint32_t led, const struct *led_info* **info)

Optional API callback to get LED information.

See also:

led_get_info() for argument descriptions.

typedef int (*`led_api_set_brightness`)(const struct *device* *dev, uint32_t led, uint8_t value)

Callback API for setting brightness of an LED.

See also:

led_set_brightness() for argument descriptions.

typedef int (*`led_api_set_color`)(const struct *device* *dev, uint32_t led, uint8_t num_colors, const uint8_t *color)

Optional API callback to set the colors of a LED.

See also:

led_set_color() for argument descriptions.

typedef int (*`led_api_on`)(const struct *device* *dev, uint32_t led)

Callback API for turning on an LED.

See also:

led_on() for argument descriptions.

typedef int (*`led_api_off`)(const struct *device* *dev, uint32_t led)

Callback API for turning off an LED.

See also:

led_off() for argument descriptions.

typedef int (*`led_api_write_channels`)(const struct *device* *dev, uint32_t start_channel, uint32_t num_channels, const uint8_t *buf)

Callback API for writing a strip of LED channels.

See also:

led_api_write_channels() for arguments descriptions.

Functions

int `led_blink`(const struct *device* *dev, uint32_t led, uint32_t delay_on, uint32_t delay_off)

Blink an LED.

This optional routine starts blinking a LED forever with the given time period.

Parameters

- `dev` – LED device
- `led` – LED number
- `delay_on` – Time period (in milliseconds) an LED should be ON
- `delay_off` – Time period (in milliseconds) an LED should be OFF

Returns 0 on success, negative on error

int `led_get_info`(const struct *device* *dev, uint32_t led, const struct *led_info* **info)

Get LED information.

This optional routine provides information about a LED.

Parameters

- `dev` – LED device
- `led` – LED number
- `info` – Pointer to a pointer filled with LED information

Returns 0 on success, negative on error

int `led_set_brightness`(const struct *device* *dev, uint32_t led, uint8_t value)

Set LED brightness.

This optional routine sets the brightness of a LED to the given value. Calling this function after *led_blink()* won't affect blinking.

LEDs which can only be turned on or off may provide this function. These should simply turn the LED on if `value` is nonzero, and off if `value` is zero.

Parameters

- `dev` – LED device
- `led` – LED number
- `value` – Brightness value to set in percent

Returns 0 on success, negative on error

int `led_write_channels`(const struct *device* *dev, uint32_t start_channel, uint32_t num_channels, const uint8_t *buf)

Write/update a strip of LED channels.

This optional routine writes a strip of LED channels to the given array of levels. Therefore it can be used to configure several LEDs at the same time.

Calling this function after *led_blink()* won't affect blinking.

Parameters

- `dev` – LED device
- `start_channel` – Absolute number (i.e. not relative to a LED) of the first channel to update.
- `num_channels` – The number of channels to write/update.
- `buf` – array of values to configure the channels with. num_channels entries must be provided.

Returns 0 on success, negative on error

int `led_set_channel`(const struct *device* *dev, uint32_t channel, uint8_t value)

Set a single LED channel.

This optional routine sets a single LED channel to the given value.

Calling this function after *led_blink()* won't affect blinking.

Parameters

- `dev` – LED device
- `channel` – Absolute channel number (i.e. not relative to a LED)
- `value` – Value to configure the channel with

Returns 0 on success, negative on error

int `led_set_color`(const struct *device* *dev, uint32_t led, uint8_t num_colors, const uint8_t *color)

Set LED color.

This routine configures all the color channels of a LED with the given color array.

Calling this function after *led_blink()* won't affect blinking.

Parameters

- `dev` – LED device
- `led` – LED number
- `num_colors` – Number of colors in the array.
- `color` – Array of colors. It must be ordered following the color mapping of the LED controller. See the the color_mapping member in struct *led_info*.

Returns 0 on success, negative on error

int `led_on`(const struct *device* *dev, uint32_t led)

Turn on an LED.

This routine turns on an LED

Parameters

- `dev` – LED device
- `led` – LED number

Returns 0 on success, negative on error

int `led_off`(const struct *device* *dev, uint32_t led)

Turn off an LED.

This routine turns off an LED

Parameters

- `dev` – LED device

- `led` – LED number

Returns 0 on success, negative on error

struct `led_info`

#include <led.h> LED information structure.

This structure gathers useful information about LED controller.

Param label LED label.

Param num_colors Number of colors per LED.

Param index Index of the LED on the controller.

Param color_mapping Mapping of the LED colors.

struct `led_driver_api`

#include <led.h> LED driver API.

LED Strip

group `led_strip_interface`

LED Strip Interface.

Typedefs

typedef int (*`led_api_update_rgb`)(const struct *device* *dev, struct *led_rgb* *pixels, size_t num_pixels)

Callback API for updating an RGB LED strip.

See also:

led_strip_update_rgb() for argument descriptions.

typedef int (*`led_api_update_channels`)(const struct *device* *dev, uint8_t *channels, size_t num_channels)

Callback API for updating channels without an RGB interpretation.

See also:

led_strip_update_channels() for argument descriptions.

Functions

static inline int `led_strip_update_rgb`(const struct *device* *dev, struct *led_rgb* *pixels, size_t num_pixels)

Update an LED strip made of RGB pixels.

Important: This routine may overwrite *pixels*.

This routine immediately updates the strip display according to the given pixels array.

Warning: May overwrite *pixels*

Parameters

- `dev` – LED strip device
- `pixels` – Array of pixel data
- `num_pixels` – Length of pixels array

Returns 0 on success, negative on error

static inline int `led_strip_update_channels`(const struct *device* *dev, uint8_t *channels, size_t num_channels)

Update an LED strip on a per-channel basis.

Important: This routine may overwrite *channels*.

This routine immediately updates the strip display according to the given channels array. Each channel byte corresponds to an individually addressable color channel or LED. Channels are updated linearly in strip order.

Warning: May overwrite *channels*

Parameters

- `dev` – LED strip device
- `channels` – Array of per-channel data
- `num_channels` – Length of channels array

Returns 0 on success, negative on error

struct `led_rgb`

#include <led_strip.h> Color value for a single RGB LED.

Individual strip drivers may ignore lower-order bits if their resolution in any channel is less than a full byte.

Public Members

uint8_t `r`

Red channel

uint8_t `g`

Green channel

uint8_t `b`

Blue channel

struct `led_strip_driver_api`

#include <led_strip.h> LED strip driver API.

This is the mandatory API any LED strip driver needs to expose.

7.21.18 Pinmux

Overview

API Reference

group `pinmux_interface`

Pinmux Interface.

Defines

`PINMUX_FUNC_A`

`PINMUX_FUNC_B`

`PINMUX_FUNC_C`

`PINMUX_FUNC_D`

`PINMUX_FUNC_E`

`PINMUX_FUNC_F`

`PINMUX_FUNC_G`

`PINMUX_FUNC_H`

`PINMUX_FUNC_I`

`PINMUX_FUNC_J`

`PINMUX_FUNC_K`

`PINMUX_FUNC_L`

`PINMUX_FUNC_M`

`PINMUX_FUNC_N`

`PINMUX_FUNC_O`

`PINMUX_FUNC_P`

`PINMUX_PULLUP_ENABLE`

PINMUX_PULLUP_DISABLE

PINMUX_INPUT_ENABLED

PINMUX_OUTPUT_ENABLED

Typedefs

typedef int (*pmux_set)(const struct *device* *dev, uint32_t pin, uint32_t func)

Callback API upon setting a PIN's function See *pinmux_pin_set()* for argument description.

typedef int (*pmux_get)(const struct *device* *dev, uint32_t pin, uint32_t *func)

Callback API upon getting a PIN's function See *pinmux_pin_get()* for argument description.

typedef int (*pmux_pullup)(const struct *device* *dev, uint32_t pin, uint8_t func)

Callback API upon setting a PIN's pullup See pinmix_pin_pullup() for argument description.

typedef int (*pmux_input)(const struct *device* *dev, uint32_t pin, uint8_t func)

Callback API upon setting a PIN's input function See pinmux_input() for argument description.

Functions

static inline int pinmux_pin_set(const struct *device* *dev, uint32_t pin, uint32_t func)

static inline int pinmux_pin_get(const struct *device* *dev, uint32_t pin, uint32_t *func)

static inline int pinmux_pin_pullup(const struct *device* *dev, uint32_t pin, uint8_t func)

static inline int pinmux_pin_input_enable(const struct *device* *dev, uint32_t pin, uint8_t func)

struct pinmux_driver_api

#include <pinmux.h>

7.21.19 PWM

Overview

API Reference

group pwm_interface

PWM Interface.

PWM capture configuration flags

PWM_CAPTURE_TYPE_PERIOD

PWM pin capture captures period.

PWM_CAPTURE_TYPE_PULSE

PWM pin capture captures pulse width.

PWM_CAPTURE_TYPE_BOTH

PWM pin capture captures both period and pulse width.

PWM_CAPTURE_MODE_SINGLE

PWM pin capture captures a single period/pulse width.

PWM_CAPTURE_MODE_CONTINUOUS

PWM pin capture captures period/pulse width continuously.

Typedefs

typedef uint8_t pwm_flags_t

Provides a type to hold PWM configuration flags.

typedef int (*pwm_pin_set_t)(const struct *device* *dev, uint32_t pwm, uint32_t period_cycles, uint32_t pulse_cycles, *pwm_flags_t* flags)

Callback API upon setting the pin See pwm_pin_set_cycles() for argument description.

typedef void (*pwm_capture_callback_handler_t)(const struct *device* *dev, uint32_t pwm, uint32_t period_cycles, uint32_t pulse_cycles, int status, void *user_data)

PWM capture callback handler function signature.

Note: The callback handler will be called in interrupt context.

Note: CONFIG_PWM_CAPTURE must be selected to enable PWM capture support.

Param dev Pointer to the device structure for the driver instance.

Param pwm PWM pin.

Param period_cycles Captured PWM period width (in clock cycles). HW specific.

Param pulse_cycles Captured PWM pulse width (in clock cycles). HW specific.

Param status Status for the PWM capture (0 if no error, negative errno otherwise. See pwm_pin_capture_cycles() return value descriptions for details).

Param user_data User data passed to pwm_pin_configure_capture()

typedef int (*pwm_pin_configure_capture_t)(const struct *device* *dev, uint32_t pwm, *pwm_flags_t* flags, *pwm_capture_callback_handler_t* cb, void *user_data)

Callback API upon configuring PWM pin capture See pwm_pin_configure_capture() for argument description.

typedef int (*pwm_pin_enable_capture_t)(const struct *device* *dev, uint32_t pwm)

Callback API upon enabling PWM pin capture See pwm_pin_enable_capture() for argument description.

typedef int (*pwm_pin_disable_capture_t)(const struct *device* *dev, uint32_t pwm)

Callback API upon disabling PWM pin capture See pwm_pin_disable_capture() for argument description.

typedef int (*pwm_get_cycles_per_sec_t)(const struct *device* *dev, uint32_t pwm, uint64_t *cycles)

Callback API upon getting cycles per second See pwm_get_cycles_per_sec() for argument description.

Functions

int pwm_pin_set_cycles(const struct *device* *dev, uint32_t pwm, uint32_t period, uint32_t pulse, *pwm_flags_t* flags)

Set the period and pulse width for a single PWM output.

The PWM period and pulse width will synchronously be set to the new values without glitches in the PWM signal, but the call will not block for the change to take effect.

Passing 0 as `pulse` will cause the pin to be driven to a constant inactive level. Passing a non-zero `pulse` equal to `period` will cause the pin to be driven to a constant active level.

Note: Not all PWM controllers support synchronous, glitch-free updates of the PWM period and pulse width. Depending on the hardware, changing the PWM period and/or pulse width may cause a glitch in the generated PWM signal.

Note: Some multi-channel PWM controllers share the PWM period across all channels. Depending on the hardware, changing the PWM period for one channel may affect the PWM period for the other channels of the same PWM controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pwm` – PWM pin.
- `period` – Period (in clock cycle) set to the PWM. HW specific.
- `pulse` – Pulse width (in clock cycle) set to the PWM. HW specific.
- `flags` – Flags for pin configuration (polarity).

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

static inline int pwm_pin_configure_capture(const struct *device* *dev, uint32_t pwm, *pwm_flags_t* flags, *pwm_capture_callback_handler_t* cb, void *user_data)

Configure PWM period/pulse width capture for a single PWM input.

After configuring PWM capture using this function, the capture can be enabled/disabled using pwm_pin_enable_capture() and pwm_pin_disable_capture().

Note: This API function cannot be invoked from user space due to the use of a function callback. In user space, one of the simpler API functions (pwm_pin_capture_cycles(), pwm_pin_capture_usec(), or pwm_pin_capture_nsec()) can be used instead.

Note: CONFIG_PWM_CAPTURE must be selected for this function to be available.

Parameters

- dev – Pointer to the device structure for the driver instance.
- pwm – PWM pin.
- flags – PWM capture flags
- cb – Application callback handler function to be called upon capture
- user_data – User data to pass to the application callback handler function

Return values

- -EINVAL – if invalid function parameters were given
- -ENOSYS – if PWM capture is not supported or the given flags are not supported
- -EIO – if IO error occurred while configuring
- -EBUSY – if PWM capture is already in progress

int pwm_pin_enable_capture(const struct *device* *dev, uint32_t pwm)

Enable PWM period/pulse width capture for a single PWM input.

The PWM pin must be configured using pwm_pin_configure_capture() prior to calling this function.

Note: CONFIG_PWM_CAPTURE must be selected for this function to be available.

Parameters

- dev – Pointer to the device structure for the driver instance.
- pwm – PWM pin.

Return values

- 0 – If successful.
- -EINVAL – if invalid function parameters were given
- -ENOSYS – if PWM capture is not supported
- -EIO – if IO error occurred while enabling PWM capture
- -EBUSY – if PWM capture is already in progress

int pwm_pin_disable_capture(const struct *device* *dev, uint32_t pwm)

Disable PWM period/pulse width capture for a single PWM input.

Note: CONFIG_PWM_CAPTURE must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pwm` – PWM pin.

Return values

- `0` – If successful.
- `-EINVAL` – if invalid function parameters were given
- `-ENOSYS` – if PWM capture is not supported
- `-EIO` – if IO error occurred while disabling PWM capture

int `pwm_pin_capture_cycles`(const struct *device* *dev, uint32_t pwm, *pwm_flags_t* flags, uint32_t *period, uint32_t *pulse, *k_timeout_t* timeout)

Capture a single PWM period/pulse width in clock cycles for a single PWM input.

This API function wraps calls to pwm_pin_configure_capture(), pwm_pin_enable_capture(), and pwm_pin_disable_capture() and passes the capture result to the caller. The function is blocking until either the PWM capture is completed or a timeout occurs.

Note: `CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pwm` – PWM pin.
- `flags` – PWM capture flags.
- `period` – Pointer to the memory to store the captured PWM period width (in clock cycles). HW specific.
- `pulse` – Pointer to the memory to store the captured PWM pulse width (in clock cycles). HW specific.
- `timeout` – Waiting period for the capture to complete.

Return values

- `0` – If successful.
- `-EBUSY` – PWM capture already in progress.
- `-EAGAIN` – Waiting period timed out.
- `-EIO` – IO error while capturing.
- `-ERANGE` – If result is too large.

int `pwm_get_cycles_per_sec`(const struct *device* *dev, uint32_t pwm, uint64_t *cycles)

Get the clock rate (cycles per second) for a single PWM output.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pwm` – PWM pin.
- `cycles` – Pointer to the memory to store clock rate (cycles per sec). HW specific.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

static inline int pwm_pin_set_usec(const struct *device* *dev, uint32_t pwm, uint32_t period, uint32_t pulse, *pwm_flags_t* flags)

Set the period and pulse width for a single PWM output.

Parameters

- dev – Pointer to the device structure for the driver instance.
- pwm – PWM pin.
- period – Period (in microseconds) set to the PWM.
- pulse – Pulse width (in microseconds) set to the PWM.
- flags – Flags for pin configuration (polarity).

Return values

- 0 – If successful.
- Negative – errno code if failure.

static inline int pwm_pin_set_nsec(const struct *device* *dev, uint32_t pwm, uint32_t period, uint32_t pulse, *pwm_flags_t* flags)

Set the period and pulse width for a single PWM output.

Parameters

- dev – Pointer to the device structure for the driver instance.
- pwm – PWM pin.
- period – Period (in nanoseconds) set to the PWM.
- pulse – Pulse width (in nanoseconds) set to the PWM.
- flags – Flags for pin configuration (polarity).

Return values

- 0 – If successful.
- Negative – errno code if failure.

static inline int pwm_pin_cycles_to_usec(const struct *device* *dev, uint32_t pwm, uint32_t cycles, uint64_t *usec)

Convert from PWM cycles to microseconds.

Parameters

- dev – Pointer to the device structure for the driver instance.
- pwm – PWM pin.
- cycles – Cycles to be converted.
- usec – Pointer to the memory to store calculated usec.

Return values

- 0 – If successful.
- -EIO – If cycles per second cannot be determined.
- -ERANGE – If result is too large.

static inline int pwm_pin_cycles_to_nsec(const struct *device* *dev, uint32_t pwm, uint32_t cycles, uint64_t *nsec)

Convert from PWM cycles to nanoseconds.

Parameters

- dev – Pointer to the device structure for the driver instance.

- `pwm` – PWM pin.
- `cycles` – Cycles to be converted.
- `nsec` – Pointer to the memory to store the calculated nsec.

Return values

- `0` – If successful.
- `-EIO` – If cycles per second cannot be determined.
- `-ERANGE` – If result is too large.

static inline int `pwm_pin_capture_usec`(const struct *device* *dev, uint32_t pwm, *pwm_flags_t* flags, uint64_t *period, uint64_t *pulse, *k_timeout_t* timeout)

Capture a single PWM period/pulse width in microseconds for a single PWM input.

This API function wraps calls to pwm_pin_capture_cycles() and pwm_pin_cycles_to_usec() and passes the capture result to the caller. The function is blocking until either the PWM capture is completed or a timeout occurs.

Note: `CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pwm` – PWM pin.
- `flags` – PWM capture flags.
- `period` – Pointer to the memory to store the captured PWM period width (in usec).
- `pulse` – Pointer to the memory to store the captured PWM pulse width (in usec).
- `timeout` – Waiting period for the capture to complete.

Return values

- `0` – If successful.
- `-EBUSY` – PWM capture already in progress.
- `-EAGAIN` – Waiting period timed out.
- `-EIO` – IO error while capturing.
- `-ERANGE` – If result is too large.

static inline int `pwm_pin_capture_nsec`(const struct *device* *dev, uint32_t pwm, *pwm_flags_t* flags, uint64_t *period, uint64_t *pulse, *k_timeout_t* timeout)

Capture a single PWM period/pulse width in nanoseconds for a single PWM input.

This API function wraps calls to pwm_pin_capture_cycles() and pwm_pin_cycles_to_nsec() and passes the capture result to the caller. The function is blocking until either the PWM capture is completed or a timeout occurs.

Note: `CONFIG_PWM_CAPTURE` must be selected for this function to be available.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pwm` – PWM pin.
- `flags` – PWM capture flags.
- `period` – Pointer to the memory to store the captured PWM period width (in nsec).
- `pulse` – Pointer to the memory to store the captured PWM pulse width (in nsec).
- `timeout` – Waiting period for the capture to complete.

Return values

- `0` – If successful.
- `-EBUSY` – PWM capture already in progress.
- `-EAGAIN` – Waiting period timed out.
- `-EIO` – IO error while capturing.
- `-ERANGE` – If result is too large.

struct `pwm_driver_api`

#include <pwm.h> PWM driver API definition.

7.21.20 PS/2

Overview

The PS/2 connector first hit the market in 1987 on IBM's desktop PC line of the same name before becoming an industry-wide standard for mouse and keyboard connections. Starting around 2007, USB superseded PS/2 and is the modern peripheral device connection standard. For legacy support on boards with a PS/2 connector, Zephyr provides these PS/2 driver APIs.

Configuration Options

Related configuration options:

- `CONFIG_PS2`

API Reference

group `ps2_interface`

PS/2 Driver APIs.

Typedefs

typedef void (*`ps2_callback_t`)(const struct *device* *dev, uint8_t data)

PS/2 callback called when user types or click a mouse.

Param dev Pointer to the device structure for the driver instance.

Param data Data byte passed pack to the user.

Functions

int `ps2_config`(const struct *device* *dev, *ps2_callback_t* callback_isr)

Configure a ps2 instance.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback_isr` – called when PS/2 devices reply to a configuration command or when a mouse/keyboard send data to the client application.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_write`(const struct *device* *dev, uint8_t value)

Write to PS/2 device.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `value` – Data for the PS2 device.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_read`(const struct *device* *dev, uint8_t *value)

Read slave-to-host values from PS/2 device.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `value` – Pointer used for reading the PS/2 device.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_enable_callback`(const struct *device* *dev)

Enables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `ps2_disable_callback`(const struct *device* *dev)

Disables callback.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

7.21.21 PECI

Overview

The Platform Environment Control Interface, abbreviated as PECI, is a thermal management standard introduced in 2006 with the Intel Core 2 Duo Microprocessors. The PECI interface allows external devices to read processor temperature, perform processor manageability functions, and manage processor interface tuning and diagnostics. The PECI bus driver APIs enable the interaction between Embedded Microcontrollers and CPUs.

Configuration Options

Related configuration options:

- `CONFIG_PECI`

API Reference

group `peci_interface`

PECI Interface 3.0.

Defines

`PECI_CC_RSP_SUCCESS`

PECI read/write supported responses

`PECI_CC_RSP_TIMEOUT`

`PECI_CC_OUT_OF_RESOURCES_TIMEOUT`

`PECI_CC_RESOURCES_LOWPWR_TIMEOUT`

`PECI_CC_ILLEGAL_REQUEST`

`PECI_PING_WR_LEN`

Ping command format.

`PECI_PING_RD_LEN`

`PECI_PING_LEN`

`PECI_GET_DIB_WR_LEN`

GetDIB command format.

`PECI_GET_DIB_RD_LEN`

`PECI_GET_DIB_CMD_LEN`

PECI_GET_DIB_DEVINFO

PECI_GET_DIB_REVNUM

PECI_GET_DIB_DOMAIN_BIT_MASK

PECI_GET_DIB_MAJOR_REV_MASK

PECI_GET_DIB_MINOR_REV_MASK

PECI_GET_TEMP_WR_LEN
: GetTemp command format.

PECI_GET_TEMP_RD_LEN

PECI_GET_TEMP_CMD_LEN

PECI_GET_TEMP_LSB

PECI_GET_TEMP_MSB

PECI_GET_TEMP_ERR_MSB

PECI_GET_TEMP_ERR_LSB_GENERAL

PECI_GET_TEMP_ERR_LSB_RES

PECI_GET_TEMP_ERR_LSB_TEMP_LO

PECI_GET_TEMP_ERR_LSB_TEMP_HI

PECI_RD_PKG_WR_LEN
: RdPkgConfig command format.

PECI_RD_PKG_LEN_BYTE

PECI_RD_PKG_LEN_WORD

PECI_RD_PKG_LEN_DWORD

PECI_RD_PKG_CMD_LEN

PECI_WR_PKG_RD_LEN
: WrPkgConfig command format

PECI_WR_PKG_LEN_BYTE

PECI_WR_PKG_LEN_WORD

PECI_WR_PKG_LEN_DWORD

PECI_WR_PKG_CMD_LEN

PECI_RD_IAMSR_WR_LEN
: RdIAMSR command format

PECI_RD_IAMSR_LEN_BYTE

PECI_RD_IAMSR_LEN_WORD

PECI_RD_IAMSR_LEN_DWORD

PECI_RD_IAMSR_LEN_QWORD

PECI_RD_IAMSR_CMD_LEN

PECI_WR_IAMSR_RD_LEN
: WrIAMSR command format

PECI_WR_IAMSR_LEN_BYTE

PECI_WR_IAMSR_LEN_WORD

PECI_WR_IAMSR_LEN_DWORD

PECI_WR_IAMSR_LEN_QWORD

PECI_WR_IAMSR_CMD_LEN

PECI_RD_PCICFG_WR_LEN
: RdPCIConfig command format

PECI_RD_PCICFG_LEN_BYTE

PECI_RD_PCICFG_LEN_WORD

PECI_RD_PCICFG_LEN_DWORD

PECI_RD_PCICFG_CMD_LEN

PECI_WR_PCICFG_RD_LEN
: WrPCIConfig command format

PECI_WR_PCICFG_LEN_BYTE

PECI_WR_PCICFG_LEN_WORD

PECI_WR_PCICFG_LEN_DWORD

PECI_WR_PCICFG_CMD_LEN

PECI_RD_PCICFGL_WR_LEN
RdPCIConfigLocal command format

PECI_RD_PCICFGL_RD_LEN_BYTE

PECI_RD_PCICFGL_RD_LEN_WORD

PECI_RD_PCICFGL_RD_LEN_DWORD

PECI_RD_PCICFGL_CMD_LEN

PECI_WR_PCICFGL_RD_LEN
WrPCIConfigLocal command format

PECI_WR_PCICFGL_WR_LEN_BYTE

PECI_WR_PCICFGL_WR_LEN_WORD

PECI_WR_PCICFGL_WR_LEN_DWORD

PECI_WR_PCICFGL_CMD_LEN

Enums

enum peci_error_code
PECI error codes.
Values:

enumerator PECI_GENERAL_SENSOR_ERROR = 0x8000

enumerator PECI_UNDERFLOW_SENSOR_ERROR = 0x8002

enumerator PECI_OVERFLOW_SENSOR_ERROR = 0x8003

enum peci_command_code
PECI commands.
Values:

enumerator PECI_CMD_PING = 0x00

enumerator PECI_CMD_GET_TEMP0 = 0x01

enumerator PECI_CMD_GET_TEMP1 = 0x02

enumerator PECI_CMD_RD_PCI_CFG0 = 0x61

enumerator PECI_CMD_RD_PCI_CFG1 = 0x62

enumerator PECI_CMD_WR_PCI_CFG0 = 0x65

enumerator PECI_CMD_WR_PCI_CFG1 = 0x66

enumerator PECI_CMD_RD_PKG_CFG0 = 0xA1

enumerator PECI_CMD_RD_PKG_CFG1 = 0xA

enumerator PECI_CMD_WR_PKG_CFG0 = 0xA5

enumerator PECI_CMD_WR_PKG_CFG1 = 0xA6

enumerator PECI_CMD_RD_IAMSR0 = 0xB1

enumerator PECI_CMD_RD_IAMSR1 = 0xB2

enumerator PECI_CMD_WR_IAMSR0 = 0xB5

enumerator PECI_CMD_WR_IAMSR1 = 0xB6

enumerator PECI_CMD_RD_PCI_CFG_LOCAL0 = 0xE1

enumerator PECI_CMD_RD_PCI_CFG_LOCAL1 = 0xE2

enumerator PECI_CMD_WR_PCI_CFG_LOCAL0 = 0xE5

enumerator PECI_CMD_WR_PCI_CFG_LOCAL1 = 0xE6

enumerator PECI_CMD_GET_DIB = 0xF7

Functions

int peci_config(const struct *device* *dev, uint32_t bitrate)

Configures the PECI interface.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `bitrate` – the selected expressed in Kbps. command or when an event needs to be sent to the client application.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `peci_enable`(const struct *device* *dev)

Enable PECI interface.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `peci_disable`(const struct *device* *dev)

Disable PECI interface.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

int `peci_transfer`(const struct *device* *dev, struct *peci_msg* *msg)

Performs a PECI transaction.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `msg` – Structure representing a PECI transaction.

Return values

- `0` – If successful.
- `Negative` – errno code if failure.

struct `peci_buf`

#include <peci.h> PECI buffer structure.

Note: Frame check sequence byte is added into rx buffer, need to allocate an additional byte for this in rx buffer.

Param buf is a valid pointer on a data buffer, or NULL otherwise.

Param len is the length of the data buffer expected to received without considering the frame check sequence byte.

struct `peci_msg`

#include <peci.h> PECI transaction packet format.

Public Members

uint8_t `addr`

Client address

enum *peci_command_code* `cmd_code`

Command code

struct *peci_buf* `tx_buffer`

Pointer to buffer of write data

struct *peci_buf* `rx_buffer`

Pointer to buffer of read data

uint8_t `flags`

PECI msg flags

7.21.22 Regulators

This subsystem provides control of voltage and current regulators. A common example is a GPIO that controls a transistor that supplies current to a device that is not always needed.

Conceptually regulators have two modes: off and on. A transition between modes may involve a time delay, so operations on regulators are inherently asynchronous. To maximize flexibility the *On-Off Manager* infrastructure is used in the generic API for the regulator subsystem. Nodes with a devicetree compatible of `regulator-fixed` are the most common flexible regulators.

In some cases the transitions are close enough to instantaneous that the the asynchronous driver implementation is not needed, and the resource cost in RAM is not justified. Such a regulator still uses the asynchronous API, but may be implemented internally in a way that ensures the result of the operation is presented before the transition completes. Zephyr recognizes devicetree nodes with a compatible of `regulator-fixed-sync` as devices with synchronous transitions.

The `vin-supply` devicetree property is used to identify the regulator(s) that a devicetree node directly depends on. Within the driver for the node the regulator API is used to issue requests for power when the device is to be active, and release the power request when the device shuts down.

The simplest case where a regulator is needed is one where there is only one client. For those situations the cost of using even the optimized synchronous regulator device infrastructure is not justified, and the `supply-gpios` devicetree property should be used. There is no device interface to these regulators as they are entirely controlled within the driver for the corresponding node, e.g. a sensor.

API Reference

group `regulator_interface`

Regulator Interface.

Functions

static inline int `regulator_enable`(const struct *device* *reg, struct *onoff_client* *cli)

Enable a regulator.

Reference-counted request that a regulator be turned on. This is an asynchronous operation; if successfully initiated the result will be communicated through the `cli` parameter.

A regulator is considered "on" when it has reached a stable/usable state.

Note: This function is *isr-ok* and *pre-kernel-ok*.

Parameters

- `reg` – a regulator device
- `cli` – used to notify the caller when the attempt to turn on the regulator has completed.

Returns non-negative on successful initiation of the request. Negative values indicate failures from *onoff_request()* or individual regulator drivers.

static inline int `regulator_disable`(const struct *device* *reg)

Disable a regulator.

Release a regulator after a previous *regulator_enable()* completed successfully.

If the release removes the last dependency on the regulator it will begin a transition to its "off" state. There is currently no mechanism to notify when the regulator has completely turned off.

This must be invoked at most once for each successful *regulator_enable()*.

Note: This function is *isr-ok*.

Parameters

- `reg` – a regulator device

Returns non-negative on successful completion of the release request. Negative values indicate failures from *onoff_release()* or individual regulator drivers.

struct `regulator_driver_api`

#include <regulator.h> Driver-specific API functions to support regulator control.

7.21.23 RTC

Overview

This is a placeholder for API specific to real-time clocks. Currently all RTC peripherals are implemented through *Counter* with device-specific API for counters with real-time support.

API Reference

group `rtc_interface`

RTC DS3231 Driver-Specific API.

Typedefs

typedef void (*`maxim_ds3231_alarm_callback_handler_t`)(const struct *device* *dev, uint8_t id, uint32_t syncclock, void *user_data)

Signature for DS3231 alarm callbacks.

The alarm callback is invoked from the system work queue thread. At the point the callback is invoked the corresponding alarm flags will have been cleared from the device status register. The callback is permitted to invoke operations on the device.

Param dev the device from which the callback originated

Param id the alarm id

Param syncclock the value from *maxim_ds3231_read_syncclock()* at the time the alarm interrupt was processed.

Param user_data the corresponding parameter from *maxim_ds3231_alarm::user_data*.

typedef void (*`maxim_ds3231_notify_callback`)(const struct *device* *dev, struct *sys_notify* *notify, int res)

Signature used to notify a user of the DS3231 that an asynchronous operation has completed.

Functions compatible with this type are subject to all the constraints of *sys_notify_generic_callback*.

Param dev the DS3231 device pointer

Param notify the notification structure provided in the call

Param res the result of the operation.

Functions

static inline uint32_t `maxim_ds3231_read_syncclock`(const struct *device* *dev)

Read the local synchronization clock.

Synchronization aligns the DS3231 real-time clock with a stable monotonic local clock which should have a frequency between 1 kHz and 1 MHz and be itself synchronized with the primary system time clock. The accuracy of the alignment and the maximum time between synchronization updates is affected by the resolution of this clock.

On some systems the hardware clock from k_cycles_get_32() is suitable, but on others that clock advances too quickly. The frequency of the target-specific clock is provided by *maxim_ds3231_syncclock_frequency()*.

At this time the value is captured from *`k_uptime_get_32()`*; future kernel extensions may make a higher-resolution clock available.

Note: This function is *isr-ok*.

Parameters

- `dev` – the DS3231 device pointer

Returns the current value of the synchronization clock.

static inline uint32_t `maxim_ds3231_syncclock_frequency`(const struct *device* *dev)

Get the frequency of the synchronization clock.

Provides the frequency of the clock used in *maxim_ds3231_read_syncclock()*.

Parameters

- `dev` – the DS3231 device pointer

Returns the frequency of the selected synchronization clock.

int `maxim_ds3231_ctrl_update`(const struct *device* *dev, uint8_t set_bits, uint8_t clear_bits)

Set and clear specific bits in the control register.

Note: This function assumes the device register cache is valid. It will not read the register value, and it will write to the device only if the value changes as a result of applying the set and clear changes.

Note: Unlike *maxim_ds3231_stat_update()* the return value from this function indicates the register value after changes were made. That return value is cached for use in subsequent operations.

Note: This function is *supervisor*.

Returns the non-negative updated value of the register, or a negative error code from an I2C transaction.

int `maxim_ds3231_stat_update`(const struct *device* *dev, uint8_t set_bits, uint8_t clear_bits)

Read the ctrl_stat register then set and clear bits in it.

The content of the ctrl_stat register will be read, then the set and clear bits applied and the result written back to the device (regardless of whether there appears to be a change in value).

OSF, A1F, and A2F will be written with 1s if the corresponding bits do not appear in either `set_bits` or `clear_bits`. This ensures that if any flag becomes set between the read and the write that indicator will not be cleared.

Note: Unlike *maxim_ds3231_ctrl_update()* the return value from this function indicates the register value before any changes were made.

Note: This function is *supervisor*.

Parameters

- `dev` – the DS3231 device pointer
- `set_bits` – bits to be set when writing back. Setting bits other than MAXIM_DS3231_REG_STAT_EN32kHz will have no effect.
- `clear_bits` – bits to be cleared when writing back. Include the bits for the status flags you want to clear.

Returns the non-negative register value as originally read (disregarding the effect of clears and sets), or a negative error code from an I2C transaction.

int `maxim_ds3231_get_alarm`(const struct *device* *dev, uint8_t id, struct *maxim_ds3231_alarm* *cfg)

Read a DS3231 alarm configuration.

The alarm configuration data is read from the device and reconstructed into the output parameter.

Note: This function is *supervisor*.

Parameters

- `dev` – the DS3231 device pointer.
- `id` – the alarm index, which must be 0 (for the 1 s resolution alarm) or 1 (for the 1 min resolution alarm).
- `cfg` – a pointer to a structure into which the configured alarm data will be stored.

Returns a non-negative value indicating successful conversion, or a negative error code from an I2C transaction or invalid parameter.

int `maxim_ds3231_set_alarm`(const struct *device* *dev, uint8_t id, const struct *maxim_ds3231_alarm* *cfg)

Configure a DS3231 alarm.

The alarm configuration is validated and stored into the device.

To cancel an alarm use *counter_cancel_channel_alarm()*.

Note: This function is *supervisor*.

Parameters

- `dev` – the DS3231 device pointer.
- `id` – 0 Analog to counter index. `ALARM1` is 0 and has 1 s resolution, `ALARM2` is 1 and has 1 minute resolution.
- `cfg` – a pointer to the desired alarm configuration. Both alarms are configured; if only one is to change the application must supply the existing configuration for the other.

Returns a non-negative value on success, or a negative error code from an I2C transaction or an invalid parameter.

int `maxim_ds3231_synchronize`(const struct *device* *dev, struct *sys_notify* *notify)

Synchronize the RTC against the local clock.

The RTC advances one tick per second with no access to sub-second precision. Synchronizing clocks at sub-second resolution requires enabling a 1pps signal then capturing the system clocks in a GPIO callback. This function provides that operation.

Synchronization is performed in asynchronously, and may take as long as 1 s to complete; notification of completion is provided through the `notify` parameter.

Applications should use *maxim_ds3231_get_syncpoint()* to retrieve the synchronization data collected by this operation.

Note: This function is *supervisor*.

Parameters

- `dev` – the DS3231 device pointer.
- `notify` – pointer to the object used to specify asynchronous function behavior and store completion information.

Return values

- `non-negative` – on success
- `-EBUSY` – if a synchronization or set is currently in progress
- `-EINVAL` – if notify is not provided
- `-ENOTSUP` – if the required interrupt is not configured

int `maxim_ds3231_req_syncpoint`(const struct *device* *dev, struct *k_poll_signal* *signal)

Request to update the synchronization point.

This is a variant of *maxim_ds3231_synchronize()* for use from user threads.

Parameters

- `dev` – the DS3231 device pointer.
- `signal` – pointer to a valid and ready-to-be-signalled *k_poll_signal*. May be NULL to request a synchronization point be collected without notifying when it has been updated.

Return values

- `non-negative` – on success
- `-EBUSY` – if a synchronization or set is currently in progress
- `-ENOTSUP` – if the required interrupt is not configured

int `maxim_ds3231_get_syncpoint`(const struct *device* *dev, struct *maxim_ds3231_syncpoint* *syncpoint)

Retrieve the most recent synchronization point.

This function returns the synchronization data last captured using *maxim_ds3231_synchronize()*.

Parameters

- `dev` – the DS3231 device pointer.
- `syncpoint` – where to store the synchronization data.

Return values

- `non-negative` – on success
- `-ENOENT` – if no syncpoint has been captured

int `maxim_ds3231_set`(const struct *device* *dev, const struct *maxim_ds3231_syncpoint* *syncpoint, struct *sys_notify* *notify)

Set the RTC to a time consistent with the provided synchronization.

The RTC advances one tick per second with no access to sub-second precision, and setting the clock resets the internal countdown chain. This function implements the magic necessary to set the clock while retaining as much sub-second accuracy as possible. It requires a synchronization point that pairs sub-second resolution civil time with a local synchronization clock captured at the same instant. The set operation may take as long as 1 second to complete; notification of completion is provided through the `notify` parameter.

Note: This function is *supervisor*.

Parameters

- `dev` – the DS3231 device pointer.
- `syncpoint` – the structure providing the synchronization point.
- `notify` – pointer to the object used to specify asynchronous function behavior and store completion information.

Return values

- `non-negative` – on success
- `-EINVAL` – if syncpoint or notify are null
- `-ENOTSUP` – if the required interrupt signal is not configured
- `-EBUSY` – if a synchronization or set is currently in progress

int `maxim_ds3231_check_alarms`(const struct *device* *dev)

Check for and clear flags indicating that an alarm has fired.

Returns a mask indicating alarms that are marked as having fired, and clears from stat the flags that it found set. Alarms that have been configured with a callback are not represented in the return value.

This API may be used when a persistent alarm has been programmed.

Note: This function is *supervisor*.

Parameters

- `dev` – the DS3231 device pointer.

Returns a non-negative value that may have MAXIM_DS3231_ALARM1 and/or MAXIM_DS3231_ALARM2 set, or a negative error code.

struct `maxim_ds3231_alarm`

#include <maxim_ds3231.h> Information defining the alarm configuration.

DS3231 alarms can be set to fire at specific times or at the rollover of minute, hour, day, or day of week.

When an alarm is configured with a handler an interrupt will be generated and the handler called from the system work queue.

When an alarm is configured without a handler, or a persisted alarm is present, alarms can be read using *maxim_ds3231_check_alarms()*.

Public Members

time_t `time`

Time specification for an RTC alarm.

Though specified as a UNIX time, the alarm parameters are determined by converting to civil time and interpreting the component hours, minutes, seconds, day-of-week, and day-of-month fields, mediated by the corresponding *flags*.

The year and month are ignored, but be aware that gmtime() determines day-of-week based on calendar date. Decoded alarm times will fall within 1978-01 since 1978-01-01 (first of month) was a Sunday (first of week).

maxim_ds3231_alarm_callback_handler_t `handler`

Handler to be invoked when alarms are signalled.

If this is null the alarm will not be triggered by the INTn/SQW GPIO. This is a "persisted" alarm from its role in using the DS3231 to trigger a wake from deep sleep. The application should use *maxim_ds3231_check_alarms()* to determine whether such an alarm has been triggered.

If this is not null the driver will monitor the ISW GPIO for alarm signals and will invoke the handler with a parameter carrying the value returned by *maxim_ds3231_check_alarms()*. The corresponding status flags will be cleared in the device before the handler is invoked.

The handler will be invoked from the system work queue.

void *`user_data`

User-provided pointer passed to alarm callback.

uint8_t `flags`

Flags controlling configuration of the alarm alarm.

See MAXIM_DS3231_ALARM_FLAGS_IGNSE and related constants.

Note that as described the alarm mask fields require that if a unit is not ignored, higher-precision units must also not be ignored. For example, if match on hours is enabled, match on minutes and seconds must also be enabled. Failure to comply with this requirement will cause *maxim_ds3231_set_alarm()* to return an error, leaving the alarm configuration unchanged.

struct `maxim_ds3231_syncpoint`

#include <maxim_ds3231.h> Register the RTC clock against system clocks.

This captures the same instant in both the RTC time scale and a stable system clock scale, allowing conversion between those scales.

Public Members

struct timespec `rtc`

Time from the DS3231.

This maybe in UTC, TAI, or local offset depending on how the RTC is maintained.

uint32_t `syncclock`

Value of a local clock at the same instant as *rtc*.

This is captured from a stable monotonic system clock running at between 1 kHz and 1 MHz, allowing for microsecond to millisecond accuracy in synchronization.

7.21.24 Sensors

The sensor subsystem exposes an API to uniformly access sensor devices. Common operations are: reading data and executing code when specific conditions are met.

Basic Operation

Channels Fundamentally, a channel is a quantity that a sensor device can measure.

Sensors can have multiple channels, either to represent different axes of the same physical property (e.g. acceleration); or because they can measure different properties altogether (ambient temperature, pressure and humidity). Complex sensors cover both cases, so a single device can expose three acceleration channels and a temperature one.

It is imperative that all sensors that support a given channel express results in the same unit of measurement. Consult the *API Reference* for all supported channels, along with their description and units of measurement:

Values Sensor devices return results as *sensor_value*. This representation avoids use of floating point values as they may not be supported on certain setups.

Fetching Values Getting a reading from a sensor requires two operations. First, an application instructs the driver to fetch a sample of all its channels. Then, individual channels may be read. In the case of channels with multiple axes, they can be read in a single operation by supplying the corresponding `_XYZ` channel type and a buffer of 3 *sensor_value* objects. This approach ensures consistency of channels between reads and efficiency of communication by issuing a single transaction on the underlying bus.

Below is an example illustrating the usage of the BME280 sensor, which measures ambient temperature and atmospheric pressure. Note that *sensor_sample_fetch()* is only called once, as it reads and compensates data for both channels.

```
1
2  /*
3   * Get a device structure from a devicetree node with compatible
4   * "bosch,bme280". (If there are multiple, just pick one.)
5   */
6  static const struct device *get_bme280_device(void)
7  {
8          const struct device *dev = DEVICE_DT_GET_ANY(bosch_bme280);
9
10         if (dev == NULL) {
11                 /* No such node, or the node does not have status "okay". */
12                 printk("\nError: no device found.\n");
13                 return NULL;
14         }
15
16         if (!device_is_ready(dev)) {
17                 printk("\nError: Device \"%s\" is not ready; "
18                        "check the driver initialization logs for errors.\n",
19                        dev->name);
20                 return NULL;
21         }
22
23         printk("Found device \"%s\", getting sensor data\n", dev->name);
24         return dev;
25 }
26
27 void main(void)
28 {
29         const struct device *dev = get_bme280_device();
30
31         if (dev == NULL) {
```

(continues on next page)

(continued from previous page)

```
32                  return;
33          }
34
35          while (1) {
36                  struct sensor_value temp, press, humidity;
37
38                  sensor_sample_fetch(dev);
39                  sensor_channel_get(dev, SENSOR_CHAN_AMBIENT_TEMP, &temp);
40                  sensor_channel_get(dev, SENSOR_CHAN_PRESS, &press);
41                  sensor_channel_get(dev, SENSOR_CHAN_HUMIDITY, &humidity);
42
43                  printk("temp: %d.%06d; press: %d.%06d; humidity: %d.%06d\n",
44                         temp.val1, temp.val2, press.val1, press.val2,
45                         humidity.val1, humidity.val2);
46
47                  k_sleep(K_MSEC(1000));
48          }
49  }
```

The example assumes that the returned values have type *sensor_value*, which is the case for BME280. A real application supporting multiple sensors should inspect the `type` field of the `temp` and `press` values and use the other fields of the structure accordingly.

Configuration and Attributes

Setting the communication bus and address is considered the most basic configuration for sensor devices. This setting is done at compile time, via the configuration menu. If the sensor supports interrupts, the interrupt lines and triggering parameters described below are also configured at compile time.

Alongside these communication parameters, sensor chips typically expose multiple parameters that control the accuracy and frequency of measurement. In compliance with Zephyr's design goals, most of these values are statically configured at compile time.

However, certain parameters could require runtime configuration, for example, threshold values for interrupts. These values are configured via attributes. The example in the following section showcases a sensor with an interrupt line that is triggered when the temperature crosses a threshold. The threshold is configured at runtime using an attribute.

Triggers

Triggers in Zephyr refer to the interrupt lines of the sensor chips. Many sensor chips support one or more triggers. Some examples of triggers include: new data is ready for reading, a channel value has crossed a threshold, or the device has sensed motion.

To configure a trigger, an application needs to supply a *sensor_trigger* and a handler function. The structure contains the trigger type and the channel on which the trigger must be configured.

Because most sensors are connected via SPI or I2C busses, it is not possible to communicate with them from the interrupt execution context. The execution of the trigger handler is deferred to a thread, so that data fetching operations are possible. A driver can spawn its own thread to fetch data, thus ensuring minimum latency. Alternatively, multiple sensor drivers can share a system-wide thread. The shared thread approach increases the latency of handling interrupts but uses less memory. You can configure which approach to follow for each driver. Most drivers can entirely disable triggers resulting in a smaller footprint.

The following example contains a trigger fired whenever temperature crosses the 26 degree Celsius threshold. It also samples the temperature every second. A real application would ideally disable periodic

sampling in the interest of saving power. Since the application has direct access to the kernel config symbols, no trigger is registered when triggering was disabled by the driver's configuration.

```

#define UCEL_PER_CEL 1000000
#define UCEL_PER_MCEL 1000
#define TEMP_INITIAL_CEL 25
#define TEMP_WINDOW_HALF_UCEL 500000

static const char *now_str(void)
{
        static char buf[16]; /* ...HH:MM:SS.MMM */
        uint32_t now = k_uptime_get_32();
        unsigned int ms = now % MSEC_PER_SEC;
        unsigned int s;
        unsigned int min;
        unsigned int h;

        now /= MSEC_PER_SEC;
        s = now % 60U;
        now /= 60U;
        min = now % 60U;
        now /= 60U;
        h = now;

        snprintf(buf, sizeof(buf), "%u:%02u:%02u.%03u",
                 h, min, s, ms);
        return buf;
}

#ifdef CONFIG_MCP9808_TRIGGER

static struct sensor_trigger trig;

static int set_window(const struct device *dev,
                      const struct sensor_value *temp)
{
        const int temp_ucel = temp->val1 * UCEL_PER_CEL + temp->val2;
        const int low_ucel = temp_ucel - TEMP_WINDOW_HALF_UCEL;
        const int high_ucel = temp_ucel + TEMP_WINDOW_HALF_UCEL;
        struct sensor_value val = {
                .val1 = low_ucel / UCEL_PER_CEL,
                .val2 = low_ucel % UCEL_PER_CEL,
        };
        int rc = sensor_attr_set(dev, SENSOR_CHAN_AMBIENT_TEMP,
                                 SENSOR_ATTR_LOWER_THRESH, &val);
        if (rc == 0) {
                val.val1 = high_ucel / UCEL_PER_CEL,
                val.val2 = high_ucel % UCEL_PER_CEL,
                rc = sensor_attr_set(dev, SENSOR_CHAN_AMBIENT_TEMP,
                                     SENSOR_ATTR_UPPER_THRESH, &val);
        }

        if (rc == 0) {
                printf("Alert on temp outside [%d, %d] milli-Celsius\n",
                       low_ucel / UCEL_PER_MCEL,
                       high_ucel / UCEL_PER_MCEL);
        }
```

(continues on next page)

(continued from previous page)

```
56
57          return rc;
58  }
59
60  static inline int set_window_ucel(const struct device *dev,
61                                    int temp_ucel)
62  {
63          struct sensor_value val = {
64                  .val1 = temp_ucel / UCEL_PER_CEL,
65                  .val2 = temp_ucel % UCEL_PER_CEL,
66          };
67
68          return set_window(dev, &val);
69  }
70
71  static void trigger_handler(const struct device *dev,
72                              struct sensor_trigger *trig)
73  {
74          struct sensor_value temp;
75          static size_t cnt;
76          int rc;
77
78          ++cnt;
79          rc = sensor_sample_fetch(dev);
80          if (rc != 0) {
81                  printf("sensor_sample_fetch error: %d\n", rc);
82                  return;
83          }
84          rc = sensor_channel_get(dev, SENSOR_CHAN_AMBIENT_TEMP, &temp);
85          if (rc != 0) {
86                  printf("sensor_channel_get error: %d\n", rc);
87                  return;
88          }
89
90          printf("trigger fired %u, temp %g deg C\n", cnt,
91                 sensor_value_to_double(&temp));
92          set_window(dev, &temp);
93  }
94  #endif
95
96  void main(void)
97  {
98          const struct device *dev = DEVICE_DT_GET_ANY(microchip_mcp9808);
99          int rc;
100
101         if (dev == NULL) {
102                 printf("Device not found.\n");
103                 return;
104         }
105         if (!device_is_ready(dev)) {
106                 printf("Device %s is not ready.\n", dev->name);
107                 return;
108         }
109
110 #ifdef CONFIG_MCP9808_TRIGGER
111         rc = set_window_ucel(dev, TEMP_INITIAL_CEL * UCEL_PER_CEL);
```

(continues on next page)

(continued from previous page)

```
        if (rc == 0) {
                trig.type = SENSOR_TRIG_THRESHOLD;
                trig.chan = SENSOR_CHAN_AMBIENT_TEMP;
                rc = sensor_trigger_set(dev, &trig, trigger_handler);
        }

        if (rc != 0) {
                printf("Trigger set failed: %d\n", rc);
                return;
        }
        printk("Trigger set got %d\n", rc);
#endif

        while (1) {
                struct sensor_value temp;

                rc = sensor_sample_fetch(dev);
                if (rc != 0) {
                        printf("sensor_sample_fetch error: %d\n", rc);
                        break;
                }

                rc = sensor_channel_get(dev, SENSOR_CHAN_AMBIENT_TEMP, &temp);
                if (rc != 0) {
                        printf("sensor_channel_get error: %d\n", rc);
                        break;
                }

                printf("%s: %g C\n", now_str(),
                       sensor_value_to_double(&temp));

                k_sleep(K_SECONDS(2));
        }
}
```

API Reference

group `sensor_interface`

Sensor Interface.

Defines

`SENSOR_G`

The value of gravitational constant in micro m/s ^ 2.

`SENSOR_PI`

The value of constant PI in micros.

Typedefs

typedef void (*sensor_trigger_handler_t)(const struct *device* *dev, struct *sensor_trigger* *trigger)

Callback API upon firing of a trigger.

Param dev Pointer to the sensor device

Param trigger The trigger

typedef int (*sensor_attr_set_t)(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, const struct *sensor_value* *val)

Callback API upon setting a sensor's attributes.

See *sensor_attr_set()* for argument description

typedef int (*sensor_attr_get_t)(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, struct *sensor_value* *val)

Callback API upon getting a sensor's attributes.

See *sensor_attr_get()* for argument description

typedef int (*sensor_trigger_set_t)(const struct *device* *dev, const struct *sensor_trigger* *trig, *sensor_trigger_handler_t* handler)

Callback API for setting a sensor's trigger and handler.

See *sensor_trigger_set()* for argument description

typedef int (*sensor_sample_fetch_t)(const struct *device* *dev, enum *sensor_channel* chan)

Callback API for fetching data from a sensor.

See *sensor_sample_fetch()* for argument description

typedef int (*sensor_channel_get_t)(const struct *device* *dev, enum *sensor_channel* chan, struct *sensor_value* *val)

Callback API for getting a reading from a sensor.

See *sensor_channel_get()* for argument description

Enums

enum sensor_channel

Sensor channels.

Values:

enumerator SENSOR_CHAN_ACCEL_X

Acceleration on the X axis, in m/s ^ 2.

enumerator SENSOR_CHAN_ACCEL_Y

Acceleration on the Y axis, in m/s ^ 2.

enumerator SENSOR_CHAN_ACCEL_Z

Acceleration on the Z axis, in m/s ^ 2.

enumerator SENSOR_CHAN_ACCEL_XYZ

Acceleration on the X, Y and Z axes.

enumerator SENSOR_CHAN_GYRO_X
Angular velocity around the X axis, in radians/s.

enumerator SENSOR_CHAN_GYRO_Y
Angular velocity around the Y axis, in radians/s.

enumerator SENSOR_CHAN_GYRO_Z
Angular velocity around the Z axis, in radians/s.

enumerator SENSOR_CHAN_GYRO_XYZ
Angular velocity around the X, Y and Z axes.

enumerator SENSOR_CHAN_MAGN_X
Magnetic field on the X axis, in Gauss.

enumerator SENSOR_CHAN_MAGN_Y
Magnetic field on the Y axis, in Gauss.

enumerator SENSOR_CHAN_MAGN_Z
Magnetic field on the Z axis, in Gauss.

enumerator SENSOR_CHAN_MAGN_XYZ
Magnetic field on the X, Y and Z axes.

enumerator SENSOR_CHAN_DIE_TEMP
Device die temperature in degrees Celsius.

enumerator SENSOR_CHAN_AMBIENT_TEMP
Ambient temperature in degrees Celsius.

enumerator SENSOR_CHAN_PRESS
Pressure in kilopascal.

enumerator SENSOR_CHAN_PROX
Proximity. Adimensional. A value of 1 indicates that an object is close.

enumerator SENSOR_CHAN_HUMIDITY
Humidity, in percent.

enumerator SENSOR_CHAN_LIGHT
Illuminance in visible spectrum, in lux.

enumerator SENSOR_CHAN_IR
Illuminance in infra-red spectrum, in lux.

enumerator SENSOR_CHAN_RED
Illuminance in red spectrum, in lux.

enumerator SENSOR_CHAN_GREEN
Illuminance in green spectrum, in lux.

enumerator SENSOR_CHAN_BLUE
Illuminance in blue spectrum, in lux.

enumerator SENSOR_CHAN_ALTITUDE
Altitude, in meters

enumerator SENSOR_CHAN_PM_1_0
1.0 micro-meters Particulate Matter, in ug/m ^ 3

enumerator SENSOR_CHAN_PM_2_5
2.5 micro-meters Particulate Matter, in ug/m ^ 3

enumerator SENSOR_CHAN_PM_10
10 micro-meters Particulate Matter, in ug/m ^ 3

enumerator SENSOR_CHAN_DISTANCE
Distance. From sensor to target, in meters

enumerator SENSOR_CHAN_CO2
CO2 level, in parts per million (ppm)

enumerator SENSOR_CHAN_VOC
VOC level, in parts per billion (ppb)

enumerator SENSOR_CHAN_GAS_RES
Gas sensor resistance in ohms.

enumerator SENSOR_CHAN_VOLTAGE
Voltage, in volts

enumerator SENSOR_CHAN_CURRENT
Current, in amps

enumerator SENSOR_CHAN_POWER
Power in watts

enumerator SENSOR_CHAN_RESISTANCE
Resistance , in Ohm

enumerator SENSOR_CHAN_ROTATION
Angular rotation, in degrees

enumerator SENSOR_CHAN_POS_DX
Position change on the X axis, in points.

enumerator SENSOR_CHAN_POS_DY
Position change on the Y axis, in points.

enumerator SENSOR_CHAN_POS_DZ
Position change on the Z axis, in points.

enumerator SENSOR_CHAN_RPM
Revolutions per minute, in RPM.

enumerator SENSOR_CHAN_GAUGE_VOLTAGE
Voltage, in volts

enumerator SENSOR_CHAN_GAUGE_AVG_CURRENT
Average current, in amps

enumerator SENSOR_CHAN_GAUGE_STDBY_CURRENT
Standy current, in amps

enumerator SENSOR_CHAN_GAUGE_MAX_LOAD_CURRENT
Max load current, in amps

enumerator SENSOR_CHAN_GAUGE_TEMP
Gauge temperature

enumerator SENSOR_CHAN_GAUGE_STATE_OF_CHARGE
State of charge measurement in %

enumerator SENSOR_CHAN_GAUGE_FULL_CHARGE_CAPACITY
Full Charge Capacity in mAh

enumerator SENSOR_CHAN_GAUGE_REMAINING_CHARGE_CAPACITY
Remaining Charge Capacity in mAh

enumerator SENSOR_CHAN_GAUGE_NOM_AVAIL_CAPACITY
Nominal Available Capacity in mAh

enumerator SENSOR_CHAN_GAUGE_FULL_AVAIL_CAPACITY
Full Available Capacity in mAh

enumerator SENSOR_CHAN_GAUGE_AVG_POWER
Average power in mW

enumerator SENSOR_CHAN_GAUGE_STATE_OF_HEALTH
State of health measurement in %

enumerator SENSOR_CHAN_GAUGE_TIME_TO_EMPTY
Time to empty in minutes

enumerator SENSOR_CHAN_GAUGE_TIME_TO_FULL

Time to full in minutes

enumerator SENSOR_CHAN_GAUGE_CYCLE_COUNT

Cycle count (total number of charge/discharge cycles)

enumerator SENSOR_CHAN_GAUGE_DESIGN_VOLTAGE

Design voltage of cell in V (max voltage)

enumerator SENSOR_CHAN_GAUGE_DESIRED_VOLTAGE

Desired voltage of cell in V (nominal voltage)

enumerator SENSOR_CHAN_GAUGE_DESIRED_CHARGING_CURRENT

Desired charging current in mA

enumerator SENSOR_CHAN_ALL

All channels.

enumerator SENSOR_CHAN_COMMON_COUNT

Number of all common sensor channels.

enumerator SENSOR_CHAN_PRIV_START = *SENSOR_CHAN_COMMON_COUNT*

This and higher values are sensor specific. Refer to the sensor header file.

enumerator SENSOR_CHAN_MAX = INT16_MAX

Maximum value describing a sensor channel type.

enum sensor_trigger_type

Sensor trigger types.

Values:

enumerator SENSOR_TRIG_TIMER

Timer-based trigger, useful when the sensor does not have an interrupt line.

enumerator SENSOR_TRIG_DATA_READY

Trigger fires whenever new data is ready.

enumerator SENSOR_TRIG_DELTA

Trigger fires when the selected channel varies significantly. This includes any-motion detection when the channel is acceleration or gyro. If detection is based on slope between successive channel readings, the slope threshold is configured via the *SENSOR_ATTR_SLOPE_TH* and *SENSOR_ATTR_SLOPE_DUR* attributes.

enumerator SENSOR_TRIG_NEAR_FAR

Trigger fires when a near/far event is detected.

enumerator SENSOR_TRIG_THRESHOLD

Trigger fires when channel reading transitions configured thresholds. The thresholds are configured via the *SENSOR_ATTR_LOWER_THRESH*, *SENSOR_ATTR_UPPER_THRESH*, and *SENSOR_ATTR_HYSTERESIS* attributes.

enumerator SENSOR_TRIG_TAP

Trigger fires when a single tap is detected.

enumerator SENSOR_TRIG_DOUBLE_TAP

Trigger fires when a double tap is detected.

enumerator SENSOR_TRIG_FREEFALL

Trigger fires when a free fall is detected.

enumerator SENSOR_TRIG_COMMON_COUNT

Number of all common sensor triggers.

enumerator SENSOR_TRIG_PRIV_START = *SENSOR_TRIG_COMMON_COUNT*

This and higher values are sensor specific. Refer to the sensor header file.

enumerator SENSOR_TRIG_MAX = INT16_MAX

Maximum value describing a sensor trigger type.

enum sensor_attribute

Sensor attribute types.

Values:

enumerator SENSOR_ATTR_SAMPLING_FREQUENCY

Sensor sampling frequency, i.e. how many times a second the sensor takes a measurement.

enumerator SENSOR_ATTR_LOWER_THRESH

Lower threshold for trigger.

enumerator SENSOR_ATTR_UPPER_THRESH

Upper threshold for trigger.

enumerator SENSOR_ATTR_SLOPE_TH

Threshold for any-motion (slope) trigger.

enumerator SENSOR_ATTR_SLOPE_DUR

Duration for which the slope values needs to be outside the threshold for the trigger to fire.

enumerator SENSOR_ATTR_HYSTERESIS

enumerator SENSOR_ATTR_OVERSAMPLING

Oversampling factor

enumerator SENSOR_ATTR_FULL_SCALE

Sensor range, in SI units.

enumerator SENSOR_ATTR_OFFSET

The sensor value returned will be altered by the amount indicated by offset: final_value = *sensor_value* + offset.

enumerator SENSOR_ATTR_CALIB_TARGET

Calibration target. This will be used by the internal chip's algorithms to calibrate itself on a certain axis, or all of them.

enumerator SENSOR_ATTR_CONFIGURATION

Configure the operating modes of a sensor.

enumerator SENSOR_ATTR_CALIBRATION

Set a calibration value needed by a sensor.

enumerator SENSOR_ATTR_FEATURE_MASK

Enable/disable sensor features

enumerator SENSOR_ATTR_ALERT

Alert threshold or alert enable/disable

enumerator SENSOR_ATTR_COMMON_COUNT

Number of all common sensor attributes.

enumerator SENSOR_ATTR_PRIV_START = *SENSOR_ATTR_COMMON_COUNT*

This and higher values are sensor specific. Refer to the sensor header file.

enumerator SENSOR_ATTR_MAX = INT16_MAX

Maximum value describing a sensor attribute type.

Functions

int sensor_attr_set(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, const struct *sensor_value* *val)

Set an attribute for a sensor.

Parameters

- dev – Pointer to the sensor device
- chan – The channel the attribute belongs to, if any. Some attributes may only be set for all channels of a device, depending on device capabilities.
- attr – The attribute to set
- val – The value to set the attribute to

Returns 0 if successful, negative errno code if failure.

int sensor_attr_get(const struct *device* *dev, enum *sensor_channel* chan, enum *sensor_attribute* attr, struct *sensor_value* *val)

Get an attribute for a sensor.

Parameters

- dev – Pointer to the sensor device

- `chan` – The channel the attribute belongs to, if any. Some attributes may only be set for all channels of a device, depending on device capabilities.
- `attr` – The attribute to get
- `val` – Pointer to where to store the attribute

Returns 0 if successful, negative errno code if failure.

static inline int `sensor_trigger_set`(const struct *device* *dev, struct *sensor_trigger* *trig, *sensor_trigger_handler_t* handler)

Activate a sensor's trigger and set the trigger handler.

The handler will be called from a thread, so I2C or SPI operations are safe. However, the thread's stack is limited and defined by the driver. It is currently up to the caller to ensure that the handler does not overflow the stack.

Function properties (list may not be complete) *supervisor*

Parameters

- `dev` – Pointer to the sensor device
- `trig` – The trigger to activate
- `handler` – The function that should be called when the trigger fires

Returns 0 if successful, negative errno code if failure.

int `sensor_sample_fetch`(const struct *device* *dev)

Fetch a sample from the sensor and store it in an internal driver buffer.

Read all of a sensor's active channels and, if necessary, perform any additional operations necessary to make the values useful. The user may then get individual channel values by calling *sensor_channel_get*.

Since the function communicates with the sensor device, it is unsafe to call it in an ISR if the device is connected via I2C or SPI.

Parameters

- `dev` – Pointer to the sensor device

Returns 0 if successful, negative errno code if failure.

int `sensor_sample_fetch_chan`(const struct *device* *dev, enum *sensor_channel* type)

Fetch a sample from the sensor and store it in an internal driver buffer.

Read and compute compensation for one type of sensor data (magnetometer, accelerometer, etc). The user may then get individual channel values by calling *sensor_channel_get*.

This is mostly implemented by multi function devices enabling reading at different sampling rates.

Since the function communicates with the sensor device, it is unsafe to call it in an ISR if the device is connected via I2C or SPI.

Parameters

- `dev` – Pointer to the sensor device
- `type` – The channel that needs updated

Returns 0 if successful, negative errno code if failure.

int `sensor_channel_get`(const struct *device* *dev, enum *sensor_channel* chan, struct *sensor_value* *val)

Get a reading from a sensor device.

Return a useful value for a particular channel, from the driver's internal data. Before calling this function, a sample must be obtained by calling *sensor_sample_fetch* or *sensor_sample_fetch_chan*. It is guaranteed that two subsequent calls of this function for the same channels will yield the same value, if *sensor_sample_fetch* or *sensor_sample_fetch_chan* has not been called in the meantime.

For vectorial data samples you can request all axes in just one call by passing the specific channel with _XYZ suffix. The sample will be returned at val[0], val[1] and val[2] (X, Y and Z in that order).

Parameters

- `dev` – Pointer to the sensor device
- `chan` – The channel to read
- `val` – Where to store the value

Returns 0 if successful, negative errno code if failure.

static inline int32_t `sensor_ms2_to_g`(const struct *sensor_value* *ms2)

Helper function to convert acceleration from m/s ^ 2 to Gs.

Parameters

- `ms2` – A pointer to a *sensor_value* struct holding the acceleration, in m/s ^ 2.

Returns The converted value, in Gs.

static inline void `sensor_g_to_ms2`(int32_t g, struct *sensor_value* *ms2)

Helper function to convert acceleration from Gs to m/s ^ 2.

Parameters

- `g` – The G value to be converted.
- `ms2` – A pointer to a *sensor_value* struct, where the result is stored.

static inline int32_t `sensor_rad_to_degrees`(const struct *sensor_value* *rad)

Helper function for converting radians to degrees.

Parameters

- `rad` – A pointer to a *sensor_value* struct, holding the value in radians.

Returns The converted value, in degrees.

static inline void `sensor_degrees_to_rad`(int32_t d, struct *sensor_value* *rad)

Helper function for converting degrees to radians.

Parameters

- `d` – The value (in degrees) to be converted.
- `rad` – A pointer to a *sensor_value* struct, where the result is stored.

static inline double `sensor_value_to_double`(const struct *sensor_value* *val)

Helper function for converting struct *sensor_value* to double.

Parameters

- `val` – A pointer to a *sensor_value* struct.

Returns The converted value.

static inline void sensor_value_from_double(struct *sensor_value* *val, double inp)

Helper function for converting double to struct *sensor_value*.

Parameters

- val – A pointer to a *sensor_value* struct.
- inp – The converted value.

struct sensor_value

#include <sensor.h> Representation of a sensor readout value.

The value is represented as having an integer and a fractional part, and can be obtained using the formula val1 + val2 * 10^(-6). Negative values also adhere to the above formula, but may need special attention. Here are some examples of the value representation:

```
 0.5: val1 =  0, val2 =  500000
-0.5: val1 =  0, val2 = -500000
-1.0: val1 = -1, val2 =  0
-1.5: val1 = -1, val2 = -500000
```

Public Members

int32_t val1

Integer part of the value.

int32_t val2

Fractional part of the value (in one-millionth parts).

struct sensor_trigger

#include <sensor.h> Sensor trigger spec.

Public Members

enum *sensor_trigger_type* type

Trigger type.

enum *sensor_channel* chan

Channel the trigger is set on.

struct sensor_driver_api

#include <sensor.h>

7.21.25 SPI

Overview

API Reference

group spi_interface

SPI Interface.

Defines

SPI_OP_MODE_MASTER

SPI operational mode.

SPI_OP_MODE_SLAVE

SPI_OP_MODE_MASK

SPI_OP_MODE_GET(_operation_)

SPI_MODE_CPOL

SPI Polarity & Phase Modes.

Clock Polarity: if set, clock idle state will be 1 and active state will be 0. If untouched, the inverse will be true which is the default.

SPI_MODE_CPHA

Clock Phase: this dictates when is the data captured, and depends clock's polarity. When SPI_MODE_CPOL is set and this bit as well, capture will occur on low to high transition and high to low if this bit is not set (default). This is fully reversed if CPOL is not set.

SPI_MODE_LOOP

Whatever data is transmitted is looped-back to the receiving buffer of the controller. This is fully controller dependent as some may not support this, and can be used for testing purposes only.

SPI_MODE_MASK

SPI_MODE_GET(_mode_)

SPI_TRANSFER_MSB

SPI Transfer modes (host controller dependent)

SPI_TRANSFER_LSB

SPI_WORD_SIZE_SHIFT

SPI word size.

SPI_WORD_SIZE_MASK

SPI_WORD_SIZE_GET(_operation_)

SPI_WORD_SET(_word_size_)

SPI_LINES_SINGLE

SPI MISO lines.

Some controllers support dual, quad or octal MISO lines connected to slaves. Default is single, which is the case most of the time.

SPI_LINES_DUAL

SPI_LINES_QUAD

SPI_LINES_OCTAL

SPI_LINES_MASK

SPI_HOLD_ON_CS

Specific SPI devices control bits.

SPI_LOCK_ON

SPI_CS_ACTIVE_HIGH

SPI_CS_CONTROL_PTR_DT(node_id, delay_)

Initialize and get a pointer to a *spi_cs_control* from a devicetree node identifier.

This helper is useful for initializing a device on a SPI bus. It initializes a struct *spi_cs_control* and returns a pointer to it. Here, `node_id` is a node identifier for a SPI device, not a SPI controller.

Example devicetree fragment:

```
spi@... {
        cs-gpios = <&gpio0 1 GPIO_ACTIVE_LOW>;
        spidev: spi-device@0 { ... };
};
```

Assume that `gpio0` follows the standard convention for specifying GPIOs, i.e. it has the following in its binding:

```
gpio-cells:
- pin
- flags
```

Example usage:

```
struct spi_cs_control *ctrl =
        SPI_CS_CONTROL_PTR_DT(DT_NODELABEL(spidev), 2);
```

This example is equivalent to:

```
struct spi_cs_control *ctrl =
        &(struct spi_cs_control) {
                .gpio_dev = DEVICE_DT_GET(DT_NODELABEL(gpio0)),
                .delay = 2,
                .gpio_pin = 1,
                .gpio_dt_flags = GPIO_ACTIVE_LOW
        };
```

This macro is not available in C++.

Parameters

- `node_id` – Devicetree node identifier for a device on a SPI bus
- `delay_` – The `delay` field to set in the *spi_cs_control*

Returns a pointer to the *spi_cs_control* structure

SPI_CS_CONTROL_PTR_DT_INST(inst, delay_)

Get a pointer to a *spi_cs_control* from a devicetree node.

This is equivalent to *SPI_CS_CONTROL_PTR_DT(DT_DRV_INST(inst), delay)*.

Therefore, DT_DRV_COMPAT must already be defined before using this macro.

This macro is not available in C++.

Parameters

- inst – Devicetree node instance number
- delay_ – The delay field to set in the *spi_cs_control*

Returns a pointer to the *spi_cs_control* structure

SPI_CONFIG_DT(node_id, operation_, delay_)

Structure initializer for *spi_config* from devicetree.

This helper macro expands to a static initializer for a struct *spi_config* by reading the relevant frequency, slave, and cs data from the devicetree.

Important: the cs field is initialized using *SPI_CS_CONTROL_PTR_DT()*. The gpio_dev value pointed to by this structure must be checked using *device_is_ready()* before use.

This macro is not available in C++.

Parameters

- node_id – Devicetree node identifier for the SPI device whose struct *spi_config* to create an initializer for
- operation_ – the desired operation field in the struct *spi_config*
- delay_ – the desired delay field in the struct *spi_config*'s *spi_cs_control*, if there is one

SPI_CONFIG_DT_INST(inst, operation_, delay_)

Structure initializer for *spi_config* from devicetree instance.

This is equivalent to *SPI_CONFIG_DT(DT_DRV_INST(inst), operation_, delay_)*.

This macro is not available in C++.

Parameters

- inst – Devicetree instance number
- operation_ – the desired operation field in the struct *spi_config*
- delay_ – the desired delay field in the struct *spi_config*'s *spi_cs_control*, if there is one

SPI_DT_SPEC_GET(node_id, operation_, delay_)

Structure initializer for *spi_dt_spec* from devicetree.

This helper macro expands to a static initializer for a struct *spi_dt_spec* by reading the relevant bus, frequency, slave, and cs data from the devicetree.

Important: multiple fields are automatically constructed by this macro which must be checked before use. *spi_is_ready* performs the required *device_is_ready* checks.

This macro is not available in C++.

Parameters

- node_id – Devicetree node identifier for the SPI device whose struct *spi_dt_spec* to create an initializer for
- operation_ – the desired operation field in the struct *spi_config*

- `delay_` – the desired `delay` field in the struct *spi_config*'s *spi_cs_control*, if there is one

`SPI_DT_SPEC_INST_GET(inst, operation_, delay_)`

Structure initializer for *spi_dt_spec* from devicetree instance.

This is equivalent to *`SPI_DT_SPEC_GET(DT_DRV_INST(inst), operation_, delay_)`*.

This macro is not available in C++.

Parameters

- `inst` – Devicetree instance number
- `operation_` – the desired `operation` field in the struct *spi_config*
- `delay_` – the desired `delay` field in the struct *spi_config*'s *spi_cs_control*, if there is one

Typedefs

typedef int (*`spi_api_io`)(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs)

Callback API for I/O See *spi_transceive()* for argument descriptions.

Callback API for asynchronous I/O See *spi_transceive_async()* for argument descriptions.

typedef int (*`spi_api_io_async`)(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs, struct *k_poll_signal* *async)

typedef int (*`spi_api_release`)(const struct *device* *dev, const struct *spi_config* *config)

Callback API for unlocking SPI device. See *spi_release()* for argument descriptions.

Functions

static inline bool `spi_is_ready`(const struct *spi_dt_spec* *spec)

Validate that SPI bus is ready.

Parameters

- `spec` – SPI specification from devicetree

Return values

- `true` – if the SPI bus is ready for use.
- `false` – if the SPI bus is not ready for use.

int `spi_transceive`(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs)

Read/write the specified amount of data from the SPI driver.

Note: This function is synchronous.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.

- `tx_bufs` – Buffer array where data to be sent originates from, or NULL if none.
- `rx_bufs` – Buffer array where data to be read will be written to, or NULL if none.

Return values

- `frames` – Positive number of frames received in slave mode.
- `0` – If successful in master mode.
- `-errno` – Negative errno code on failure.

static inline int `spi_transceive_dt`(const struct *spi_dt_spec* *spec, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs)

Read/write data from an SPI bus specified in *`spi_dt_spec`*.

This is equivalent to:

```
spi_transceive(spec->bus, &spec->config, tx_bufs, rx_bufs);
```

Parameters

- `spec` – SPI specification from devicetree
- `tx_bufs` – Buffer array where data to be sent originates from, or NULL if none.
- `rx_bufs` – Buffer array where data to be read will be written to, or NULL if none.

Returns a value from *spi_transceive()*.

static inline int `spi_read`(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *rx_bufs)

Read the specified amount of data from the SPI driver.

Note: This function is synchronous.

Note: This function is an helper function calling spi_transceive.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `rx_bufs` – Buffer array where data to be read will be written to.

Return values

- `0` – If successful.
- `-errno` – Negative errno code on failure.

static inline int `spi_read_dt`(const struct *spi_dt_spec* *spec, const struct *spi_buf_set* *rx_bufs)

Read data from a SPI bus specified in *`spi_dt_spec`*.

This is equivalent to:

```
spi_read(spec->bus, &spec->config, rx_bufs);
```

Parameters

- `spec` – SPI specification from devicetree
- `rx_bufs` – Buffer array where data to be read will be written to.

Returns a value from *spi_read()*.

static inline int `spi_write`(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs)

Write the specified amount of data from the SPI driver.

Note: This function is synchronous.

Note: This function is an helper function calling spi_transceive.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `tx_bufs` – Buffer array where data to be sent originates from.

Return values

- `0` – If successful.
- `-errno` – Negative errno code on failure.

static inline int `spi_write_dt`(const struct *spi_dt_spec* *spec, const struct *spi_buf_set* *tx_bufs)

Write data to a SPI bus specified in *spi_dt_spec*.

This is equivalent to:

```
spi_write(spec->bus, &spec->config, tx_bufs);
```

Parameters

- `spec` – SPI specification from devicetree
- `tx_bufs` – Buffer array where data to be sent originates from.

Returns a value from *spi_write()*.

static inline int `spi_transceive_async`(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, const struct *spi_buf_set* *rx_bufs, struct *k_poll_signal* *async)

Read/write the specified amount of data from the SPI driver.

Note: This function is asynchronous.

Note: This function is available only if `CONFIG_SPI_ASYNC` is selected.

Parameters

- `dev` – Pointer to the device structure for the driver instance

- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `tx_bufs` – Buffer array where data to be sent originates from, or NULL if none.
- `rx_bufs` – Buffer array where data to be read will be written to, or NULL if none.
- `async` – A pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Return values

- `frames` – Positive number of frames received in slave mode.
- `0` – If successful in master mode.
- `-errno` – Negative errno code on failure.

static inline int `spi_read_async`(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *rx_bufs, struct *k_poll_signal* *async)

Read the specified amount of data from the SPI driver.

Note: This function is asynchronous.

Note: This function is an helper function calling spi_transceive_async.

Note: This function is available only if `CONFIG_SPI_ASYNC` is selected.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `rx_bufs` – Buffer array where data to be read will be written to.
- `async` – A pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Return values

- `0` – If successful
- `-errno` – Negative errno code on failure.

static inline int `spi_write_async`(const struct *device* *dev, const struct *spi_config* *config, const struct *spi_buf_set* *tx_bufs, struct *k_poll_signal* *async)

Write the specified amount of data from the SPI driver.

Note: This function is asynchronous.

Note: This function is an helper function calling spi_transceive_async.

Note: This function is available only if `CONFIG_SPI_ASYNC` is selected.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.
- `tx_bufs` – Buffer array where data to be sent originates from.
- `async` – A pointer to a valid and ready to be signaled struct *k_poll_signal*. (Note: if NULL this function will not notify the end of the transaction, and whether it went successfully or not).

Return values

- `0` – If successful.
- `-errno` – Negative errno code on failure.

int `spi_release`(const struct *device* *dev, const struct *spi_config* *config)

Release the SPI device locked on by the current config.

Note: This synchronous function is used to release the lock on the SPI device that was kept if, and if only, given config parameter was the last one to be used (in any of the above functions) and if it has the SPI_LOCK_ON bit set into its operation bits field. This can be used if the caller needs to keep its hand on the SPI device for consecutive transactions.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `config` – Pointer to a valid *spi_config* structure instance. Pointer-comparison may be used to detect changes from previous operations.

Return values

- `0` – If successful.
- `-errno` – Negative errno code on failure.

static inline int `spi_release_dt`(const struct *spi_dt_spec* *spec)

Release the SPI device specified in *spi_dt_spec*.

This is equivalent to:

```
spi_release(spec->bus, &spec->config);
```

Parameters

- `spec` – SPI specification from devicetree

Returns a value from *spi_release()*.

struct `spi_cs_control`

#include <spi.h> SPI Chip Select control structure.

This can be used to control a CS line via a GPIO line, instead of using the controller inner CS logic.

Param gpio_dev is a valid pointer to an actual GPIO device. A NULL pointer can be provided to full inhibit CS control if necessary.

Param gpio_pin is a number representing the gpio PIN that will be used to act as a CS line

Param delay is a delay in microseconds to wait before starting the transmission and before releasing the CS line

Param gpio_dt_flags is the devicetree flags corresponding to how the CS line should be driven. GPIO_ACTIVE_LOW/GPIO_ACTIVE_HIGH should be equivalent to SPI_CS_ACTIVE_HIGH/SPI_CS_ACTIVE_LOW options in struct *spi_config*.

struct `spi_config`

#include <spi.h> SPI controller configuration structure.

Note: Only cs_hold and lock_on can be changed between consecutive transceive call. Rest of the attributes are not meant to be tweaked.

Warning: Most drivers use pointer comparison to determine whether a passed configuration is different from one used in a previous transaction. Changes to fields in the structure may not be detected.

Param frequency is the bus frequency in Hertz

Param operation is a bit field with the following parts:

```
operational mode    [ 0 ]       - master or slave.
mode                [ 1 : 3 ]   - Polarity, phase and loop mode.
transfer            [ 4 ]       - LSB or MSB first.
word_size           [ 5 : 10 ]  - Size of a data frame in bits.
lines               [ 11 : 12 ] - MISO lines: Single/Dual/Quad/
↪Octal.
cs_hold             [ 13 ]      - Hold on the CS line if possible.
lock_on             [ 14 ]      - Keep resource locked for the␣
↪caller.
cs_active_high      [ 15 ]      - Active high CS logic.
```

Param slave is the slave number from 0 to host controller slave limit.

Param cs is a valid pointer on a struct *spi_cs_control* is CS line is emulated through a gpio line, or NULL otherwise.

struct `spi_dt_spec`

#include <spi.h> Complete SPI DT information.

Param bus is the SPI bus

Param config is the slave specific configuration

struct `spi_buf`

#include <spi.h> SPI buffer structure.

Param buf is a valid pointer on a data buffer, or NULL otherwise.

Param len is the length of the buffer or, if buf is NULL, will be the length which as to be sent as dummy bytes (as TX buffer) or the length of bytes that should be skipped (as RX buffer).

struct `spi_buf_set`

#include <spi.h> SPI buffer array structure.

Param buffers is a valid pointer on an array of *spi_buf*, or NULL.

Param count is the length of the array pointed by buffers.

struct `spi_driver_api`

#include <spi.h> SPI driver API This is the mandatory API any SPI driver needs to expose.

7.21.26 UART

Overview

API Reference

group `uart_interface`

UART Interface.

Typedefs

typedef void (*`uart_callback_t`)(const struct *device* *dev, struct *uart_event* *evt, void *user_data)

Define the application callback function signature for *uart_callback_set()* function.

Param dev UART device structure.

Param evt Pointer to *uart_event* structure.

Param user_data Pointer to data specified by user.

typedef void (*`uart_irq_callback_user_data_t`)(const struct *device* *dev, void *user_data)

Define the application callback function signature for *uart_irq_callback_user_data_set()* function.

Param dev UART device structure.

Param user_data Arbitrary user data.

typedef void (*`uart_irq_config_func_t`)(const struct *device* *dev)

For configuring IRQ on each individual UART device.

Param dev UART device structure.

Enums

enum `uart_line_ctrl`

Line control signals.

Values:

enumerator `UART_LINE_CTRL_BAUD_RATE` = *BIT*(0)

enumerator UART_LINE_CTRL_RTS = *BIT*(1)

enumerator UART_LINE_CTRL_DTR = *BIT*(2)

enumerator UART_LINE_CTRL_DCD = *BIT*(3)

enumerator UART_LINE_CTRL_DSR = *BIT*(4)

enum uart_event_type

Types of events passed to callback in UART_ASYNC_API.

Receiving:

a. To start receiving, uart_rx_enable has to be called with first buffer

b. When receiving starts to current buffer, *uart_event_type::UART_RX_BUF_REQUEST* will be generated, in response to that user can either:

- Provide second buffer using uart_rx_buf_rsp, when first buffer is filled, receiving will automatically start to second buffer.
- Ignore the event, this way when current buffer is filled *uart_event_type::UART_RX_RDY* event will be generated and receiving will be stopped.

c. If some data was received and timeout occurred *uart_event_type::UART_RX_RDY* event will be generated. It can happen multiples times for the same buffer. RX timeout is counted from last byte received i.e. if no data was received, there won't be any timeout event.

d. After buffer is filled *uart_event_type::UART_RX_RDY* will be generated, immediately followed by *uart_event_type::UART_RX_BUF_RELEASED* indicating that current buffer is no longer used.

e. If there was second buffer provided, it will become current buffer and we start again at point 2. If no second buffer was specified receiving is stopped and *uart_event_type::UART_RX_DISABLED* event is generated. After that whole process can be repeated.

Any time during reception *uart_event_type::UART_RX_STOPPED* event can occur. if there is any data received, *uart_event_type::UART_RX_RDY* event will be generated. It will be followed by *uart_event_type::UART_RX_BUF_RELEASED* event for every buffer currently passed to driver and finally by *uart_event_type::UART_RX_DISABLED* event.

Receiving can be disabled using uart_rx_disable, after calling that function, if there is any data received, *uart_event_type::UART_RX_RDY* event will be generated. *uart_event_type::UART_RX_BUF_RELEASED* event will be generated for every buffer currently passed to driver and finally *uart_event_type::UART_RX_DISABLED* event will occur.

Transmitting:

a. Transmitting starts by uart_tx function.

b. If whole buffer was transmitted *uart_event_type::UART_TX_DONE* is generated. If timeout occurred *uart_event_type::UART_TX_ABORTED* will be generated.

Transmitting can be aborted using *uart_tx_abort*, after calling that function *uart_event_type::UART_TX_ABORTED* event will be generated.

Values:

enumerator UART_TX_DONE

Whole TX buffer was transmitted.

enumerator UART_TX_ABORTED

Transmitting aborted due to timeout or uart_tx_abort call.

When flow control is enabled, there is a possibility that TX transfer won't finish in the allotted time. Some data may have been transferred, information about it can be found in event data.

enumerator UART_RX_RDY

Received data is ready for processing.

This event is generated in the following cases:

- When RX timeout occurred, and data was stored in provided buffer. This can happen multiple times in the same buffer.
- When provided buffer is full.
- After *uart_rx_disable()*.
- After stopping due to external event (*uart_event_type::UART_RX_STOPPED*).

enumerator UART_RX_BUF_REQUEST

Driver requests next buffer for continuous reception.

This event is triggered when receiving has started for a new buffer, i.e. it's time to provide a next buffer for a seamless switchover to it. For continuous reliable receiving, user should provide another RX buffer in response to this event, using uart_rx_buf_rsp function

If uart_rx_buf_rsp is not called before current buffer is filled up, receiving will stop.

enumerator UART_RX_BUF_RELEASED

Buffer is no longer used by UART driver.

enumerator UART_RX_DISABLED

RX has been disabled and can be reenabled.

This event is generated whenever receiver has been stopped, disabled or finished its operation and can be enabled again using uart_rx_enable

enumerator UART_RX_STOPPED

RX has stopped due to external event.

Reason is one of uart_rx_stop_reason.

enum uart_rx_stop_reason

Reception stop reasons.

Values that correspond to events or errors responsible for stopping receiving.

Values:

enumerator UART_ERROR_OVERRUN = (1 << 0)

Overrun error.

enumerator UART_ERROR_PARITY = (1 << 1)

Parity error.

enumerator UART_ERROR_FRAMING = (1 << 2)

Framing error.

enumerator UART_BREAK = (1 << 3)

Break interrupt.

A break interrupt was received. This happens when the serial input is held at a logic '0' state for longer than the sum of start time + data bits + parity + stop bits.

enumerator UART_ERROR_COLLISION = (1 << 4)

Collision error.

This error is raised when transmitted data does not match received data. Typically this is useful in scenarios where the TX and RX lines maybe connected together such as RS-485 half-duplex. This error is only valid on UARTs that support collision checking.

enum uart_config_parity

Parity modes.

Values:

enumerator UART_CFG_PARITY_NONE

enumerator UART_CFG_PARITY_ODD

enumerator UART_CFG_PARITY_EVEN

enumerator UART_CFG_PARITY_MARK

enumerator UART_CFG_PARITY_SPACE

enum uart_config_stop_bits

Number of stop bits.

Values:

enumerator UART_CFG_STOP_BITS_0_5

enumerator UART_CFG_STOP_BITS_1

enumerator UART_CFG_STOP_BITS_1_5

enumerator UART_CFG_STOP_BITS_2

enum uart_config_data_bits

Number of data bits.

Values:

enumerator UART_CFG_DATA_BITS_5

enumerator UART_CFG_DATA_BITS_6

enumerator UART_CFG_DATA_BITS_7

enumerator UART_CFG_DATA_BITS_8

enumerator UART_CFG_DATA_BITS_9

enum uart_config_flow_control

Hardware flow control options.

With flow control set to none, any operations related to flow control signals can be managed by user with uart_line_ctrl functions. In other cases, flow control is managed by hardware/driver.

Values:

enumerator UART_CFG_FLOW_CTRL_NONE

enumerator UART_CFG_FLOW_CTRL_RTS_CTS

enumerator UART_CFG_FLOW_CTRL_DTR_DSR

Functions

static inline int uart_callback_set(const struct *device* *dev, *uart_callback_t* callback, void *user_data)

Set event handler function.

Since it is mandatory to set callback to use other asynchronous functions, it can be used to detect if the device supports asynchronous API. Remaining API does not have that detection.

Parameters

- dev – UART device structure.
- callback – Event handler.
- user_data – Data to pass to event handler function.

Return values

- -ENOSYS – If not supported by the device.
- -ENOTSUP – If API not enabled.
- 0 – If successful, negative errno code otherwise.

int uart_tx(const struct *device* *dev, const uint8_t *buf, size_t len, int32_t timeout)

Send given number of bytes from buffer through UART.

Function returns immediately and event handler, set using *uart_callback_set*, is called after transfer is finished.

Parameters

- dev – UART device structure.
- buf – Pointer to transmit buffer.
- len – Length of transmit buffer.

- `timeout` – Timeout in milliseconds. Valid only if flow control is enabled. SYS_FOREVER_MS disables timeout.

Return values

- `-ENOTSUP` – If not supported.
- `-EBUSY` – There is already an ongoing transfer.
- `0` – If successful, negative errno code otherwise.

int `uart_tx_abort`(const struct *device* *dev)

Abort current TX transmission.

uart_event_type::UART_TX_DONE event will be generated with amount of data sent.

Parameters

- `dev` – UART device structure.

Return values

- `-ENOTSUP` – If not supported.
- `-EFAULT` – There is no active transmission.
- `0` – If successful, negative errno code otherwise.

int `uart_rx_enable`(const struct *device* *dev, uint8_t *buf, size_t len, int32_t timeout)

Start receiving data through UART.

Function sets given buffer as first buffer for receiving and returns immediately. After that event handler, set using *uart_callback_set*, is called with *uart_event_type::UART_RX_RDY* or *uart_event_type::UART_RX_BUF_REQUEST* events.

Parameters

- `dev` – UART device structure.
- `buf` – Pointer to receive buffer.
- `len` – Buffer length.
- `timeout` – Inactivity period after receiving at least a byte which triggers *uart_event_type::UART_RX_RDY* event. Given in milliseconds. SYS_FOREVER_MS disables timeout. See *uart_event_type* for details.

Return values

- `-ENOTSUP` – If not supported.
- `-EBUSY` – RX already in progress.
- `0` – If successful, negative errno code otherwise.

static inline int `uart_rx_buf_rsp`(const struct *device* *dev, uint8_t *buf, size_t len)

Provide receive buffer in response to *uart_event_type::UART_RX_BUF_REQUEST* event.

Provide pointer to RX buffer, which will be used when current buffer is filled.

Note: Providing buffer that is already in usage by driver leads to undefined behavior. Buffer can be reused when it has been released by driver.

Parameters

- `dev` – UART device structure.
- `buf` – Pointer to receive buffer.
- `len` – Buffer length.

Return values

- `-ENOTSUP` – If not supported.
- `-EBUSY` – Next buffer already set.
- `-EACCES` – Receiver is already disabled (function called too late?).
- `0` – If successful, negative errno code otherwise.

int `uart_rx_disable`(const struct *device* *dev)

Disable RX.

uart_event_type::UART_RX_BUF_RELEASED event will be generated for every buffer scheduled, after that *uart_event_type::UART_RX_DISABLED* event will be generated. Additionally, if there is any pending received data, the *uart_event_type::UART_RX_RDY* event for that data will be generated before the *uart_event_type::UART_RX_BUF_RELEASED* events.

Parameters

- `dev` – UART device structure.

Return values

- `-ENOTSUP` – If not supported.
- `-EFAULT` – There is no active reception.
- `0` – If successful, negative errno code otherwise.

int `uart_err_check`(const struct *device* *dev)

Check whether an error was detected.

Parameters

- `dev` – UART device structure.

Return values

- `uart_rx_stop_reason` – If error during receiving occurred.
- `0` – Otherwise.
- `-ENOSYS` – If this function is not supported.

int `uart_poll_in`(const struct *device* *dev, unsigned char *p_char)

Poll the device for input.

Parameters

- `dev` – UART device structure.
- `p_char` – Pointer to character.

Return values

- `0` – If a character arrived.
- `-1` – If no character was available to read (i.e., the UART input buffer was empty).
- `-ENOSYS` – If the operation is not supported.
- `-EBUSY` – If reception was enabled using uart_rx_enabled

void `uart_poll_out`(const struct *device* *dev, unsigned char out_char)

Output a character in polled mode.

This routine checks if the transmitter is empty. When the transmitter is empty, it writes a character to the data register.

To send a character when hardware flow control is enabled, the handshake signal CTS must be asserted.

Parameters

- `dev` – UART device structure.
- `out_char` – Character to send.

int `uart_configure`(const struct *device* *dev, const struct *uart_config* *cfg)

Set UART configuration.

Sets UART configuration using data from *cfg.

Parameters

- `dev` – UART device structure.
- `cfg` – UART configuration structure.

Return values

- `-ENOSYS` – If configuration is not supported by device. or driver does not support setting configuration in runtime.
- `0` – If successful, negative errno code otherwise.

int `uart_config_get`(const struct *device* *dev, struct *uart_config* *cfg)

Get UART configuration.

Stores current UART configuration to *cfg, can be used to retrieve initial configuration after device was initialized using data from DTS.

Parameters

- `dev` – UART device structure.
- `cfg` – UART configuration structure.

Return values

- `-ENOSYS` – If driver does not support getting current configuration.
- `0` – If successful, negative errno code otherwise.

static inline int `uart_fifo_fill`(const struct *device* *dev, const uint8_t *tx_data, int size)

Fill FIFO with data.

This function is expected to be called from UART interrupt handler (ISR), if *uart_irq_tx_ready()* returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). Likewise, *not* calling this function from an ISR if *uart_irq_tx_ready()* returns true may lead to undefined behavior, e.g. infinite interrupt loops. It's mandatory to test return value of this function, as different hardware has different FIFO depth (oftentimes just 1).

Parameters

- `dev` – UART device structure.
- `tx_data` – Data to transmit.
- `size` – Number of bytes to send.

Return values

- `-ENOSYS` – if this function is not supported
- `-ENOTSUP` – if API is not enabled.

Returns Number of bytes sent.

static inline int `uart_fifo_read`(const struct *device* *dev, uint8_t *rx_data, const int size)

Read data from FIFO.

This function is expected to be called from UART interrupt handler (ISR), if *uart_irq_rx_ready()* returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). It's unspecified whether "RX ready" condition as returned by *uart_irq_rx_ready()* is level- or edge- triggered. That means that once *uart_irq_rx_ready()* is detected, *uart_fifo_read()* must be called until it reads all available data in the FIFO (i.e. until it returns less data than was requested).

Note that the calling context only applies to physical UARTs and no to the virtual ones found in USB CDC ACM code.

Parameters

- `dev` – UART device structure.
- `rx_data` – Data container.
- `size` – Container size.

Return values

- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.

Returns Number of bytes read.

void `uart_irq_tx_enable`(const struct *device* *dev)

Enable TX interrupt in IER.

Parameters

- `dev` – UART device structure.

Returns N/A

void `uart_irq_tx_disable`(const struct *device* *dev)

Disable TX interrupt in IER.

Parameters

- `dev` – UART device structure.

Returns N/A

static inline int `uart_irq_tx_ready`(const struct *device* *dev)

Check if UART TX buffer can accept a new char.

Check if UART TX buffer can accept at least one character for transmission (i.e. *uart_fifo_fill()* will succeed and return non-zero). This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, *uart_irq_update()* must be called once per the handler invocation.

Parameters

- `dev` – UART device structure.

Return values

- `1` – If at least one char can be written to UART.
- `0` – If device is not ready to write a new byte.
- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.

void `uart_irq_rx_enable`(const struct *device* *dev)

Enable RX interrupt.

Parameters

- `dev` – UART device structure.

Returns N/A

void `uart_irq_rx_disable`(const struct *device* *dev)

Disable RX interrupt.

Parameters

- `dev` – UART device structure.

Returns N/A

static inline int `uart_irq_tx_complete`(const struct *device* *dev)

Check if UART TX block finished transmission.

Check if any outgoing data buffered in UART TX block was fully transmitted and TX block is idle. When this condition is true, UART device (or whole system) can be power off. Note that this function is *not* useful to check if UART TX can accept more data, use *uart_irq_tx_ready()* for that. This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, *uart_irq_update()* must be called once per the handler invocation.

Parameters

- `dev` – UART device structure.

Return values

- `1` – If nothing remains to be transmitted.
- `0` – If transmission is not completed.
- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.

static inline int `uart_irq_rx_ready`(const struct *device* *dev)

Check if UART RX buffer has a received char.

Check if UART RX buffer has at least one pending character (i.e. *uart_fifo_read()* will succeed and return non-zero). This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, *uart_irq_update()* must be called once per the handler invocation. It's unspecified whether condition as returned by this function is level- or edge- triggered (i.e. if this function returns true when RX FIFO is non-empty, or when a new char was received since last call to it). See description of *uart_fifo_read()* for implication of this.

Parameters

- `dev` – UART device structure.

Return values

- `1` – If a received char is ready.
- `0` – If a received char is not ready.
- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.

void `uart_irq_err_enable`(const struct *device* *dev)

Enable error interrupt.

Parameters

- `dev` – UART device structure.

Returns N/A

void `uart_irq_err_disable`(const struct *device* *dev)

Disable error interrupt.

Parameters

- `dev` – UART device structure.

Return values

- `1` – If an IRQ is ready.
- `0` – Otherwise.

int `uart_irq_is_pending`(const struct *device* *dev)

Check if any IRQs is pending.

Parameters

- `dev` – UART device structure.

Return values

- `1` – If an IRQ is pending.
- `0` – If an IRQ is not pending.
- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.

int `uart_irq_update`(const struct *device* *dev)

Start processing interrupts in ISR.

This function should be called the first thing in the ISR. Calling *uart_irq_rx_ready()*, *uart_irq_tx_ready()*, *uart_irq_tx_complete()* allowed only after this.

The purpose of this function is:

- For devices with auto-acknowledge of interrupt status on register read to cache the value of this register (rx_ready, etc. then use this case).
- For devices with explicit acknowledgement of interrupts, to ack any pending interrupts and likewise to cache the original value.
- For devices with implicit acknowledgement, this function will be empty. But the ISR must perform the actions needs to ack the interrupts (usually, call *uart_fifo_read()* on rx_ready, and *uart_fifo_fill()* on tx_ready).

Parameters

- `dev` – UART device structure.

Return values

- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.
- `1` – On success.

static inline void `uart_irq_callback_user_data_set`(const struct *device* *dev, *uart_irq_callback_user_data_t* cb, void *user_data)

Set the IRQ callback function pointer.

This sets up the callback for IRQ. When an IRQ is triggered, the specified function will be called with specified user data. See description of *uart_irq_update()* for the requirements on ISR.

Parameters

- `dev` – UART device structure.
- `cb` – Pointer to the callback function.
- `user_data` – Data to pass to callback function.

Returns N/A

static inline void `uart_irq_callback_set`(const struct *device* *dev, *uart_irq_callback_user_data_t* cb)

Set the IRQ callback function pointer (legacy).

This sets up the callback for IRQ. When an IRQ is triggered, the specified function will be called with the device pointer.

Parameters

- `dev` – UART device structure.
- `cb` – Pointer to the callback function.

Returns N/A

int `uart_line_ctrl_set`(const struct *device* *dev, uint32_t ctrl, uint32_t val)

Manipulate line control for UART.

Parameters

- `dev` – UART device structure.
- `ctrl` – The line control to manipulate (see enum uart_line_ctrl).
- `val` – Value to set to the line control.

Return values

- `0` – If successful.
- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.
- `negative` – value if failed.

int `uart_line_ctrl_get`(const struct *device* *dev, uint32_t ctrl, uint32_t *val)

Retrieve line control for UART.

Parameters

- `dev` – UART device structure.
- `ctrl` – The line control to retrieve (see enum uart_line_ctrl).
- `val` – Pointer to variable where to store the line control value.

Return values

- `0` – If successful.
- `-ENOSYS` – if this function is not supported.

- `-ENOTSUP` – if API is not enabled.
- `negative` – value if failed.

int `uart_drv_cmd`(const struct *device* *dev, uint32_t cmd, uint32_t p)

Send extra command to driver.

Implementation and accepted commands are driver specific. Refer to the drivers for more information.

Parameters

- `dev` – UART device structure.
- `cmd` – Command to driver.
- `p` – Parameter to the command.

Return values

- `0` – If successful.
- `-ENOSYS` – if this function is not supported.
- `-ENOTSUP` – if API is not enabled.
- `negative` – value if failed.

struct `uart_event_tx`

#include <uart.h> UART TX event data.

Public Members

const uint8_t *`buf`

Pointer to current buffer.

size_t `len`

Number of bytes sent.

struct `uart_event_rx`

#include <uart.h> UART RX event data.

The data represented by the event is stored in rx.buf[rx.offset] to rx.buf[rx.offset+rx.len]. That is, the length is relative to the offset.

Public Members

uint8_t *`buf`

Pointer to current buffer.

size_t `offset`

Currently received data offset in bytes.

size_t `len`

Number of new bytes received.

struct `uart_event_rx_buf`

#include <uart.h> UART RX buffer released event data.

struct `uart_event_rx_stop`

#include <uart.h> UART RX stopped data.

Public Members

enum *uart_rx_stop_reason* `reason`

Reason why receiving stopped.

struct *uart_event_rx* `data`

Last received data.

struct `uart_event`

#include <uart.h> Structure containing information about current event.

Public Members

enum *uart_event_type* `type`

Type of event.

union `uart_event_data`

#include <uart.h> Event data.

Public Members

struct *uart_event_tx* `tx`

uart_event_type::UART_TX_DONE and *uart_event_type::UART_TX_ABORTED* events data.

struct *uart_event_rx* `rx`

uart_event_type::UART_RX_RDY event data.

struct *uart_event_rx_buf* `rx_buf`

uart_event_type::UART_RX_BUF_RELEASED event data.

struct *uart_event_rx_stop* `rx_stop`

uart_event_type::UART_RX_STOPPED event data.

struct `uart_config`

#include <uart.h> UART controller configuration structure.

Param baudrate Baudrate setting in bps

Param parity Parity bit, use *uart_config_parity*

Param stop_bits Stop bits, use *uart_config_stop_bits*

Param data_bits Data bits, use *uart_config_data_bits*

Param flow_ctrl Flow control setting, use *uart_config_flow_control*

struct `uart_device_config`

#include <uart.h> UART device configuration.

Param port Base port number

Param base Memory mapped base address

Param regs Register address

Param sys_clk_freq System clock frequency in Hz

struct `uart_driver_api`

#include <uart.h> Driver API structure.

Public Members

int (*`poll_in`)(const struct *device* *dev, unsigned char *p_char)

Console I/O function

int (*`err_check`)(const struct *device* *dev)

Console I/O function

int (*`configure`)(const struct *device* *dev, const struct *uart_config* *cfg)

UART configuration functions

int (*`fifo_fill`)(const struct *device* *dev, const uint8_t *tx_data, int len)

Interrupt driven FIFO fill function

int (*`fifo_read`)(const struct *device* *dev, uint8_t *rx_data, const int size)

Interrupt driven FIFO read function

void (*`irq_tx_enable`)(const struct *device* *dev)

Interrupt driven transfer enabling function

void (*`irq_tx_disable`)(const struct *device* *dev)

Interrupt driven transfer disabling function

int (*`irq_tx_ready`)(const struct *device* *dev)

Interrupt driven transfer ready function

void (*`irq_rx_enable`)(const struct *device* *dev)

Interrupt driven receiver enabling function

void (*`irq_rx_disable`)(const struct *device* *dev)

Interrupt driven receiver disabling function

int (*`irq_tx_complete`)(const struct *device* *dev)

Interrupt driven transfer complete function

int (*irq_rx_ready)(const struct *device* *dev)
Interrupt driven receiver ready function

void (*irq_err_enable)(const struct *device* *dev)
Interrupt driven error enabling function

void (*irq_err_disable)(const struct *device* *dev)
Interrupt driven error disabling function

int (*irq_is_pending)(const struct *device* *dev)
Interrupt driven pending status function

int (*irq_update)(const struct *device* *dev)
Interrupt driven interrupt update function

void (*irq_callback_set)(const struct *device* *dev, *uart_irq_callback_user_data_t* cb, void *user_data)
Set the irq callback function

7.21.27 MDIO

Overview

MDIO is a bus that is commonly used to communicate with ethernet PHY devices. Many ethernet MAC controllers also provide hardware to communicate over MDIO bus with a peripheral device.

This API is intended to be used primarily by PHY drivers but can also be used by user firmware.

API Reference

group mdio_interface
MDIO Interface.

Functions

void mdio_bus_enable(const struct *device* *dev)
Enable MDIO bus.

Parameters

- dev – **[in]** Pointer to the device structure for the controller

void mdio_bus_disable(const struct *device* *dev)
Disable MDIO bus and tri-state drivers.

Parameters

- dev – **[in]** Pointer to the device structure for the controller

int mdio_read(const struct *device* *dev, uint8_t prtad, uint8_t devad, uint16_t *data)
Read from MDIO Bus.

This routine provides a generic interface to perform a read on the MDIO bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller
- `prtad` – **[in]** Port address
- `devad` – **[in]** Device address
- `data` – Pointer to receive read data

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ETIMEDOUT` – If transaction timedout on the bus

int `mdio_write`(const struct *device* *dev, uint8_t prtad, uint8_t devad, uint16_t data)

Write to MDIO bus.

This routine provides a generic interface to perform a write on the MDIO bus.

Parameters

- `dev` – **[in]** Pointer to the device structure for the controller
- `prtad` – **[in]** Port address
- `devad` – **[in]** Device address
- `data` – **[in]** Data to write

Return values

- `0` – If successful.
- `-EIO` – General input / output error.
- `-ETIMEDOUT` – If transaction timedout on the bus

7.21.28 Watchdog

Overview

API Reference

group `watchdog_interface`

Watchdog Interface.

Watchdog Reset Behavior.

Reset behavior after timeout.

`WDT_FLAG_RESET_NONE`

No reset

`WDT_FLAG_RESET_CPU_CORE`

CPU core reset

`WDT_FLAG_RESET_SOC`

Global SoC reset

Defines

WDT_OPT_PAUSE_IN_SLEEP

Pause watchdog timer when CPU is in sleep state.

WDT_OPT_PAUSE_HALTED_BY_DBG

Pause watchdog timer when CPU is halted by the debugger.

WDT_FLAG_RESET_SHIFT

Watchdog reset flag bit field mask shift.

WDT_FLAG_RESET_MASK

Watchdog reset flag bit field mask.

Typedefs

typedef void (*wdt_callback_t)(const struct *device* *dev, int channel_id)

Watchdog callback.

typedef int (*wdt_api_setup)(const struct *device* *dev, uint8_t options)

Callback API for setting up watchdog instance. See *wdt_setup()* for argument descriptions.

typedef int (*wdt_api_disable)(const struct *device* *dev)

Callback API for disabling watchdog instance. See *wdt_disable()* for argument descriptions.

typedef int (*wdt_api_install_timeout)(const struct *device* *dev, const struct *wdt_timeout_cfg* *cfg)

Callback API for installing new timeout. See *wdt_install_timeout()* for argument descriptions.

typedef int (*wdt_api_feed)(const struct *device* *dev, int channel_id)

Callback API for feeding specified watchdog timeout. See (wdt_feed) for argument descriptions.

Functions

int wdt_setup(const struct *device* *dev, uint8_t options)

Set up watchdog instance.

This function is used for configuring global watchdog settings that affect all timeouts. It should be called after installing timeouts. After successful return, all installed timeouts are valid and must be serviced periodically by calling *wdt_feed()*.

Parameters

- dev – Pointer to the device structure for the driver instance.
- options – Configuration options as defined by the WDT_OPT_* constants

Return values

- 0 – If successful.
- -ENOTSUP – If any of the set options is not supported.

- -EBUSY – If watchdog instance has been already setup.

int wdt_disable(const struct *device* *dev)

Disable watchdog instance.

This function disables the watchdog instance and automatically uninstalls all timeouts. To set up a new watchdog, install timeouts and call *wdt_setup()* again. Not all watchdogs can be restarted after they are disabled.

Parameters

- dev – Pointer to the device structure for the driver instance.

Return values

- 0 – If successful.
- -EFAULT – If watchdog instance is not enabled.
- -EPERM – If watchdog can not be disabled directly by application code.

static inline int wdt_install_timeout(const struct *device* *dev, const struct *wdt_timeout_cfg* *cfg)

Install new timeout.

This function must be used before *wdt_setup()*. Changes applied here have no effects until *wdt_setup()* is called.

Parameters

- dev – Pointer to the device structure for the driver instance.
- cfg – Pointer to timeout configuration structure.

Return values

- channel_id – If successful, a non-negative value indicating the index of the channel to which the timeout was assigned. This value is supposed to be used as the parameter in calls to *wdt_feed()*.
- -EBUSY – If timeout can not be installed while watchdog has already been setup.
- -ENOMEM – If no more timeouts can be installed.
- -ENOTSUP – If any of the set flags is not supported.
- -EINVAL – If any of the window timeout value is out of possible range. This value is also returned if watchdog supports only one timeout value for all timeouts and the supplied timeout window differs from windows for alarms installed so far.

int wdt_feed(const struct *device* *dev, int channel_id)

Feed specified watchdog timeout.

Parameters

- dev – Pointer to the device structure for the driver instance.
- channel_id – Index of the fed channel.

Return values

- 0 – If successful.
- -EAGAIN – If completing the feed operation would stall the caller, for example due to an in-progress watchdog operation such as a previous *wdt_feed()*.
- -EINVAL – If there is no installed timeout for supplied channel.

struct wdt_window

#include <watchdog.h> Watchdog timeout window.

Each installed timeout needs feeding within the specified time window, otherwise the watchdog will trigger. If the watchdog instance does not support window timeouts then min value must be equal to 0.

Note: If specified values can not be precisely set they are always rounded up.

Param min Lower limit of watchdog feed timeout in milliseconds.

Param max Upper limit of watchdog feed timeout in milliseconds.

struct wdt_timeout_cfg

#include <watchdog.h> Watchdog timeout configuration struct.

Param window Timing parameters of watchdog timeout.

Param callback Timeout callback. Passing NULL means that no callback will be run.

Param next Pointer to the next timeout configuration. This pointer is used for watchdogs with staged timeouts functionality. Value must be NULL for single stage timeout.

Param flags Bit field with following parts:

```
reset        [ 0 : 1 ]   - perform specified reset after timeout/
↪callback
```

7.21.29 Video

The video driver API offers a generic interface to video devices.

Basic Operation

Video Device A video device is the abstraction of a hardware or software video function, which can produce, process, consume or transform video data. The video API is designed to offer flexible way to create, handle and combine various video devices.

Endpoint Each video device can have one or more endpoints. Output endpoints configure video output function and generate data. Input endpoints configure video input function and consume data.

Video Buffer A video buffer provides the transport mechanism for the data. There is no particular requirement on the content. The requirement for the content is defined by the endpoint format. A video buffer can be queued to a device endpoint for filling (input ep) or consuming (output ep) operation, once the operation is achieved, buffer can be dequeued for post-processing, release or reuse.

Controls A video control is accessed and identified by a CID (control identifier). It represents a video control property. Different devices will have different controls available which can be generic, related to a device class or vendor specific. The set/get control functions provide a generic scalable interface to handle and create controls.

Configuration Options

Related configuration options:

- CONFIG_VIDEO

API Reference

group video_interface

Video Interface.

Defines

video_fourcc(a, b, c, d)

VIDEO_PIX_FMT_BGGR8

VIDEO_PIX_FMT_GBRG8

VIDEO_PIX_FMT_GRBG8

VIDEO_PIX_FMT_RGGB8

VIDEO_PIX_FMT_RGB565

VIDEO_PIX_FMT_JPEG

Typedefs

typedef int (*video_api_set_format_t)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

Set video format See *video_set_format()* for argument descriptions.

typedef int (*video_api_get_format_t)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

get current video format See *video_get_format()* for argument descriptions.

typedef int (*video_api_enqueue_t)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* *buf)

Enqueue a buffer in the driver's incoming queue. See *video_enqueue()* for argument descriptions.

typedef int (*video_api_dequeue_t)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* **buf, *k_timeout_t* timeout)

Dequeue a buffer from the driver's outgoing queue. See *video_dequeue()* for argument descriptions.

typedef int (*video_api_flush_t)(const struct *device* *dev, enum *video_endpoint_id* ep, bool cancel)

Flush endpoint buffers, buffer are moved from incoming queue to outgoing queue. See *video_flush()* for argument descriptions.

typedef int (*video_api_stream_start_t)(const struct *device* *dev)

Start the capture or output process. See *video_stream_start()* for argument descriptions.

typedef int (*video_api_stream_stop_t)(const struct *device* *dev)

Stop the capture or output process. See *video_stream_stop()* for argument descriptions.

typedef int (*video_api_set_ctrl_t)(const struct *device* *dev, unsigned int cid, void *value)

set a video control value. See *video_set_ctrl()* for argument descriptions.

typedef int (*video_api_get_ctrl_t)(const struct *device* *dev, unsigned int cid, void *value)

get a video control value. See *video_get_ctrl()* for argument descriptions.

typedef int (*video_api_get_caps_t)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_caps* *caps)

Get capabilities of a video endpoint. See *video_get_caps()* for argument descriptions.

typedef int (*video_api_set_signal_t)(const struct *device* *dev, enum *video_endpoint_id* ep, struct *k_poll_signal* *signal)

Register/Unregister poll signal for buffer events. See *video_set_signal()* for argument descriptions.

Enums

enum video_endpoint_id

video_endpoint_id enum Identify the video device endpoint.

Values:

enumerator VIDEO_EP_NONE

enumerator VIDEO_EP_ANY

enumerator VIDEO_EP_IN

enumerator VIDEO_EP_OUT

enum video_signal_result

video_event enum Identify video event.

Values:

enumerator VIDEO_BUF_DONE

enumerator VIDEO_BUF_ABORTED

enumerator VIDEO_BUF_ERROR

Functions

static inline int video_set_format(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

Set video format.

Configure video device with a specific format.

Parameters

- dev – Pointer to the device structure for the driver instance.
- ep – Endpoint ID.
- fmt – Pointer to a video format struct.

Return values

- 0 – Is successful.
- -EINVAL – If parameters are invalid.
- -ENOTSUP – If format is not supported.
- -EIO – General input / output error.

static inline int video_get_format(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_format* *fmt)

Get video format.

Get video device current video format.

Parameters

- dev – Pointer to the device structure for the driver instance.
- ep – Endpoint ID.
- fmt – Pointer to video format struct.

Return values pointer – to video format

static inline int video_enqueue(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* *buf)

Enqueue a video buffer.

Enqueue an empty (capturing) or filled (output) video buffer in the driver's endpoint incoming queue.

Parameters

- dev – Pointer to the device structure for the driver instance.
- ep – Endpoint ID.
- buf – Pointer to the video buffer.

Return values

- 0 – Is successful.
- -EINVAL – If parameters are invalid.
- -EIO – General input / output error.

static inline int `video_dequeue`(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_buffer* **buf, *k_timeout_t* timeout)

Dequeue a video buffer.

Dequeue a filled (capturing) or displayed (output) buffer from the driver's enpoint outgoing queue.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ep` – Endpoint ID.
- `buf` – Pointer a video buffer pointer.
- `timeout` – Timeout

Return values

- `0` – Is successful.
- `-EINVAL` – If parameters are invalid.
- `-EIO` – General input / output error.

static inline int `video_flush`(const struct *device* *dev, enum *video_endpoint_id* ep, bool cancel)

Flush endpoint buffers.

A call to flush finishes when all endpoint buffers have been moved from incoming queue to outgoing queue. Either because canceled or fully processed through the video function.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ep` – Endpoint ID.
- `cancel` – If true, cancel buffer processing instead of waiting for completion.

Return values `0` – Is successful, -ERRNO code otherwise.

static inline int `video_stream_start`(const struct *device* *dev)

Start the video device function.

video_stream_start is called to enter 'streaming' state (capture, output...). The driver may receive buffers with *video_enqueue()* before video_stream_start is called. If driver/device needs a minimum number of buffers before being able to start streaming, then driver set the min_vbuf_count to the related endpoint capabilities.

Return values

- `0` – Is successful.
- `-EIO` – General input / output error.

static inline int `video_stream_stop`(const struct *device* *dev)

Stop the video device function.

On video_stream_stop, driver must stop any transactions or wait until they finish.

Return values

- `0` – Is successful.
- `-EIO` – General input / output error.

static inline int `video_get_caps`(const struct *device* *dev, enum *video_endpoint_id* ep, struct *video_caps* *caps)

Get the capabilities of a video endpoint.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ep` – Endpoint ID.
- `caps` – Pointer to the *video_caps* struct to fill.

Return values `0` – Is successful, -ERRNO code otherwise.

static inline int `video_set_ctrl`(const struct *device* *dev, unsigned int cid, void *value)

Set the value of a control.

This set the value of a video control, value type depends on control ID, and must be interpreted accordingly.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cid` – Control ID.
- `value` – Pointer to the control value.

Return values

- `0` – Is successful.
- `-EINVAL` – If parameters are invalid.
- `-ENOTSUP` – If format is not supported.
- `-EIO` – General input / output error.

static inline int `video_get_ctrl`(const struct *device* *dev, unsigned int cid, void *value)

Get the current value of a control.

This retrieve the value of a video control, value type depends on control ID, and must be interpreted accordingly.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cid` – Control ID.
- `value` – Pointer to the control value.

Return values

- `0` – Is successful.
- `-EINVAL` – If parameters are invalid.
- `-ENOTSUP` – If format is not supported.
- `-EIO` – General input / output error.

static inline int `video_set_signal`(const struct *device* *dev, enum *video_endpoint_id* ep, struct *k_poll_signal* *signal)

Register/Unregister k_poll signal for a video endpoint.

Register a poll signal to the endpoint, which will be signaled on frame completion (done, aborted, error). Registering a NULL poll signal unregisters any previously registered signal.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ep` – Endpoint ID.
- `signal` – Pointer to *k_poll_signal*

Return values `0` – Is successful, -ERRNO code otherwise.

struct *video_buffer* *video_buffer_alloc(size_t size)

Allocate video buffer.

Parameters

- size – Size of the video buffer.

Return values pointer – to allocated video buffer

void video_buffer_release(struct *video_buffer* *buf)

Release a video buffer.

Parameters

- buf – Pointer to the video buffer to release.

struct video_format

#include <video.h> video format structure

Used to configure frame format.

Param pixelformat is the fourcc pixel format value.

Param width is the frame width in pixels.

Param height is the frame height in pixels.

Param pitch is the line stride, the number of bytes that needs to be added to the address in the first pixel of a row in order to go to the address of the first pixel of the next row (>=width).

struct video_format_cap

#include <video.h> video format capability

Used to describe a video endpoint format capability.

Param pixelformat is a list of supported pixel formats (0 terminated).

Param width_min is the minimum supported frame width.

Param width_max is the maximum supported frame width.

Param height_min is the minimum supported frame width.

Param height_max is the maximum supported frame width.

Param width_step is the width step size.

Param height_step is the height step size.

struct video_caps

#include <video.h> video capabilities

Used to describe video endpoint capabilities.

Param format_caps is a list of video format capabilities (zero terminated).

Param min_vbuf_count is the minimal count of video buffers to enqueue before being able to start the stream.

struct video_buffer

#include <video.h> video buffer structure

Represent a video frame.

Param driver_data is a pointer to driver specific data.

Param buffer is a pointer to the start of the buffer.

Param size is the size in bytes of the buffer.

Param bytesused is the number of bytes occupied by the valid data in the buffer.

Param timestamp is a time reference in milliseconds at which the last data byte was actually received for input endpoints or to be consumed for output endpoints.

struct `video_driver_api`

#include <video.h>

group `video_controls`

Video controls.

Defines

`VIDEO_CTRL_CLASS_GENERIC`

`VIDEO_CTRL_CLASS_CAMERA`

`VIDEO_CTRL_CLASS_MPEG`

`VIDEO_CTRL_CLASS_JPEG`

`VIDEO_CTRL_CLASS_VENDOR`

`VIDEO_CID_HFLIP`

`VIDEO_CID_VFLIP`

`VIDEO_CID_CAMERA_EXPOSURE`

`VIDEO_CID_CAMERA_GAIN`

`VIDEO_CID_CAMERA_ZOOM`

`VIDEO_CID_CAMERA_BRIGHTNESS`

`VIDEO_CID_CAMERA_SATURATION`

`VIDEO_CID_CAMERA_WHITE_BAL`

`VIDEO_CID_CAMERA_CONTRAST`

`VIDEO_CID_CAMERA_COLORBAR`

`VIDEO_CID_CAMERA_QUALITY`

7.21.30 eSPI

Overview

The eSPI (enhanced serial peripheral interface) is a serial bus that is based on SPI. It also features a four-wire interface (receive, transmit, clock and slave select) and three configurations: single IO, dual IO and quad IO.

The technical advancements include lower voltage signal levels (1.8V vs. 3.3V), lower pin count, and the frequency is twice as fast (66MHz vs. 33MHz) Because of its enhancements, the eSPI is used to replace the LPC (lower pin count) interface, SPI, SMBus and sideband signals.

See eSPI interface specification for additional details.

API Reference

group `espi_interface`

eSPI Driver APIs

eSPI SAF Driver APIs

Defines

`HOST_KBC_EVT_IBF`

`HOST_KBC_EVT_OBE`

Typedefs

typedef void (*`espi_callback_handler_t`)(const struct *device* *dev, struct espi_callback *cb, struct *espi_event* espi_evt)

Define the application callback handler function signature.

Param dev Device struct for the eSPI device.

Param cb Original struct espi_callback owning this handler.

Param espi_evt event details that trigger the callback handler.

Enums

enum `espi_io_mode`

eSPI I/O mode capabilities

Values:

enumerator `ESPI_IO_MODE_SINGLE_LINE` = *BIT*(0)

enumerator `ESPI_IO_MODE_DUAL_LINES` = *BIT*(1)

enumerator `ESPI_IO_MODE_QUAD_LINES` = *BIT*(2)

enum `espi_channel`

eSPI channel.

+——————————————————————-—+ | | | eSPI host +———-—+ | | +————+ | Power | +——-—+ | | |Out of band| | management | | GPIO | | | +—————+ |processor | | controller | | sources | | | | SPI flash | +————+ +————-—+ +——-—+ | | | controller | | | | | | +—————+ | | | | | | | | +———+ +——————+ | | | | | | | | | | | +–—+ +———+ +———-—+ +-—v–—+ | | | | | | LPC | | Tunneled | | Tunneled | | | | | | | bridge | | SMBus | | GPIO | | | | | | +———+ +——-—+ +——-—+ | | | | | | | | | | | | ——+ | | | | | | | | | | | | | | +——v–—+ +—v—-—v————-—v-—+ | | | | | eSPI Flash | | eSPI protocol block | | | | | | access +—>+ | | | | | +—————+ +———————————+ | | | | | | | | +————+ | | | | v v | | | XXXXXXXXXXXXXXXXXXXXXXX | | | XXXXXXXXXXXXXXXXXXXXX | | | XXXXXXXXXXXXXXXXXXX | +———————————————————————-—+ | | v +————-—+ +———+ | | | | | | Flash | | | | | | | +———+ | + + + + | eSPI bus | CH0 CH1 CH2 CH3 | (logical channels) | + + + + | | | | | | | +—————+ | +—————————————————————————+ | eSPI slave | | | | CH0 | CH1 | CH2 | CH3 | | eSPI endpoint | VWIRE | OOB | Flash | +———————————————————————-—+

Identifies each eSPI logical channel supported by eSPI controller Each channel allows independent traffic, but the assignment of channel type to channel number is fixed.

Note that generic commands are not associated with any channel, so traffic over eSPI can occur if all channels are disabled or not ready

Values:

enumerator `ESPI_CHANNEL_PERIPHERAL` = *BIT*(0)

enumerator `ESPI_CHANNEL_VWIRE` = *BIT*(1)

enumerator `ESPI_CHANNEL_OOB` = *BIT*(2)

enumerator `ESPI_CHANNEL_FLASH` = *BIT*(3)

enum `espi_bus_event`

eSPI bus event.

eSPI bus event to indicate events for which user can register callbacks

Values:

enumerator `ESPI_BUS_RESET` = *BIT*(0)

enumerator `ESPI_BUS_EVENT_CHANNEL_READY` = *BIT*(1)

enumerator `ESPI_BUS_EVENT_VWIRE_RECEIVED` = *BIT*(2)

enumerator `ESPI_BUS_EVENT_OOB_RECEIVED` = *BIT*(3)

enumerator `ESPI_BUS_PERIPHERAL_NOTIFICATION` = *BIT*(4)

enum espi_virtual_peripheral

eSPI peripheral notification type.

eSPI peripheral notification event details to indicate which peripheral trigger the eSPI callback

Values:

enumerator ESPI_PERIPHERAL_UART

enumerator ESPI_PERIPHERAL_8042_KBC

enumerator ESPI_PERIPHERAL_HOST_IO

enumerator ESPI_PERIPHERAL_DEBUG_PORT80

enumerator ESPI_PERIPHERAL_HOST_IO_PVT

enum espi_cycle_type

eSPI cycle types supported over eSPI peripheral channel

Values:

enumerator ESPI_CYCLE_MEMORY_READ32

enumerator ESPI_CYCLE_MEMORY_READ64

enumerator ESPI_CYCLE_MEMORY_WRITE32

enumerator ESPI_CYCLE_MEMORY_WRITE64

enumerator ESPI_CYCLE_MESSAGE_NODATA

enumerator ESPI_CYCLE_MESSAGE_DATA

enumerator ESPI_CYCLE_OK_COMPLETION_NODATA

enumerator ESPI_CYCLE_OKCOMPLETION_DATA

enumerator ESPI_CYCLE_NOK_COMPLETION_NODATA

enum espi_vwire_signal

eSPI system platform signals that can be send or receive through virtual wire channel

Values:

enumerator ESPI_VWIRE_SIGNAL_SLP_S3

enumerator ESPI_VWIRE_SIGNAL_SLP_S4

enumerator ESPI_VWIRE_SIGNAL_SLP_S5

enumerator ESPI_VWIRE_SIGNAL_OOB_RST_WARN

enumerator ESPI_VWIRE_SIGNAL_PLTRST

enumerator ESPI_VWIRE_SIGNAL_SUS_STAT

enumerator ESPI_VWIRE_SIGNAL_NMIOUT

enumerator ESPI_VWIRE_SIGNAL_SMIOUT

enumerator ESPI_VWIRE_SIGNAL_HOST_RST_WARN

enumerator ESPI_VWIRE_SIGNAL_SLP_A

enumerator ESPI_VWIRE_SIGNAL_SUS_PWRDN_ACK

enumerator ESPI_VWIRE_SIGNAL_SUS_WARN

enumerator ESPI_VWIRE_SIGNAL_SLP_WLAN

enumerator ESPI_VWIRE_SIGNAL_SLP_LAN

enumerator ESPI_VWIRE_SIGNAL_HOST_C10

enumerator ESPI_VWIRE_SIGNAL_DNX_WARN

enumerator ESPI_VWIRE_SIGNAL_PME

enumerator ESPI_VWIRE_SIGNAL_WAKE

enumerator ESPI_VWIRE_SIGNAL_OOB_RST_ACK

enumerator ESPI_VWIRE_SIGNAL_SLV_BOOT_STS

enumerator ESPI_VWIRE_SIGNAL_ERR_NON_FATAL

enumerator ESPI_VWIRE_SIGNAL_ERR_FATAL

enumerator ESPI_VWIRE_SIGNAL_SLV_BOOT_DONE

enumerator ESPI_VWIRE_SIGNAL_HOST_RST_ACK

enumerator ESPI_VWIRE_SIGNAL_RST_CPU_INIT

enumerator ESPI_VWIRE_SIGNAL_SMI

enumerator ESPI_VWIRE_SIGNAL_SCI

enumerator ESPI_VWIRE_SIGNAL_DNX_ACK

enumerator ESPI_VWIRE_SIGNAL_SUS_ACK

enum lpc_peripheral_opcode

Values:

enumerator E8042_OBF_HAS_CHAR = 0x50

enumerator E8042_IBF_HAS_CHAR

enumerator E8042_WRITE_KB_CHAR

enumerator E8042_WRITE_MB_CHAR

enumerator E8042_RESUME_IRQ

enumerator E8042_PAUSE_IRQ

enumerator E8042_CLEAR_OBF

enumerator E8042_READ_KB_STS

enumerator E8042_SET_FLAG

enumerator E8042_CLEAR_FLAG

enumerator EACPI_OBF_HAS_CHAR = EACPI_START_OPCODE

enumerator EACPI_IBF_HAS_CHAR

enumerator EACPI_WRITE_CHAR

enumerator EACPI_READ_STS

enumerator EACPI_WRITE_STS

Functions

int `espi_config`(const struct *device* *dev, struct *espi_cfg* *cfg)

Configure operation of a eSPI controller.

This routine provides a generic interface to override eSPI controller capabilities.

If this eSPI controller is acting as slave, the values set here will be discovered as part through the GET_CONFIGURATION command issued by the eSPI master during initialization.

If this eSPI controller is acting as master, the values set here will be used by eSPI master to determine minimum common capabilities with eSPI slave then send via SET_CONFIGURATION command.

+——+ +———+ +——+ +———+ +———+ | eSPI | | eSPI | | eSPI | | eSPI | | slave | | driver | | bus | | driver | | host | +——+ +———+ +——+ +———+ +———+ | | | | | | espi_config | Set eSPI | Set eSPI | espi_config | +—————+ ctrl regs | cap ctrl reg| +————+ | +—-—+ | +——+ |

	<——+	+—-—>
		GET_CONFIGURATION

| | +<—————+ | | |<————| | | | | eSPI caps | | | | |————>+ response | | | | |—————>+ | | | | | | | | | SET_CONFIGURATION | | | | +<—————+ | | | | accept | | | | +—————>+ |

- + + + +

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cfg` – the device runtime configuration for the eSPI controller.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.
- `-EINVAL` – invalid capabilities, failed to configure device.
- `-ENOTSUP` – capability not supported by eSPI slave.

bool `espi_get_channel_status`(const struct *device* *dev, enum *espi_channel* ch)

Query to see if it a channel is ready.

This routine allows to check if logical channel is ready before use. Note that queries for channels not supported will always return false.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `ch` – the eSPI channel for which status is to be retrieved.

Return values

- `true` – If eSPI channel is ready.
- `false` – otherwise.

int `espi_read_request`(const struct *device* *dev, struct *espi_request_packet* *req)

Sends memory, I/O or message read request over eSPI.

This routines provides a generic interface to send a read request packet.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `req` – Address of structure representing a memory, I/O or message read request.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI controller doesn't support raw packets and instead low memory transactions are handled by controller hardware directly.
- `-EIO` – General input / output error, failed to send over the bus.

int `espi_write_request`(const struct *device* *dev, struct *espi_request_packet* *req)

Sends memory, I/O or message write request over eSPI.

This routines provides a generic interface to send a write request packet.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `req` – Address of structure representing a memory, I/O or message write request.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI controller doesn't support raw packets and instead low memory transactions are handled by controller hardware directly.
- `-EINVAL` – General input / output error, failed to send over the bus.

int `espi_read_lpc_request`(const struct *device* *dev, enum *lpc_peripheral_opcode* op, uint32_t *data)

Reads SOC data from a LPC peripheral with information updated over eSPI.

This routine provides a generic interface to read a block whose information was updated by an eSPI transaction. Reading may trigger a transaction. The eSPI packet is assembled by the HW block.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `op` – Enum representing opcode for peripheral type and read request.
- `data` – Parameter to be read from to the LPC peripheral.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI peripheral is off or not supported.
- `-EINVAL` – for unimplemented lpc opcode, but in range.

int `espi_write_lpc_request`(const struct *device* *dev, enum *lpc_peripheral_opcode* op, uint32_t *data)

Writes data to a LPC peripheral which generates an eSPI transaction.

This routine provides a generic interface to write data to a block which triggers an eSPI transaction. The eSPI packet is assembled by the HW block.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `op` – Enum representing an opcode for peripheral type and write request.
- `data` – Represents the parameter passed to the LPC peripheral.

Return values

- `0` – If successful.
- `-ENOTSUP` – if eSPI peripheral is off or not supported.
- `-EINVAL` – for unimplemented lpc opcode, but in range.

int `espi_send_vwire`(const struct *device* *dev, enum *espi_vwire_signal* signal, uint8_t level)

Sends system/platform signal as a virtual wire packet.

This routines provides a generic interface to send a virtual wire packet from slave to master.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `signal` – The signal to be send to eSPI master.
- `level` – The level of signal requested LOW or HIGH.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to send over the bus.

int `espi_receive_vwire`(const struct *device* *dev, enum *espi_vwire_signal* signal, uint8_t *level)

Retrieves level status for a signal encapsulated in a virtual wire.

This routines provides a generic interface to request a virtual wire packet from eSPI master and retrieve the signal level.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `signal` – the signal to be requested from eSPI master.
- `level` – the level of signal requested 0b LOW, 1b HIGH.

Return values `-EIO` – General input / output error, failed request to master.

int `espi_send_oob`(const struct *device* *dev, struct *espi_oob_packet* *pckt)

Sends SMBus transaction (out-of-band) packet over eSPI bus.

This routines provides an interface to encapsulate a SMBus transaction and send into packet over eSPI bus

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the packet representation of SMBus transaction.

Return values `-EIO` – General input / output error, failed request to master.

int `espi_receive_oob`(const struct *device* *dev, struct *espi_oob_packet* *pckt)

Receives SMBus transaction (out-of-band) packet from eSPI bus.

This routines provides an interface to receive and decoded a SMBus transaction from eSPI bus

Parameters

- `dev` – Pointer to the device structure for the driver instance.

- `pckt` – Address of the packet representation of SMBus transaction.

Return values `-EIO` – General input / output error, failed request to master.

int `espi_read_flash`(const struct *device* *dev, struct *espi_flash_packet* *pckt)

Sends a read request packet for shared flash.

This routines provides an interface to send a request to read the flash component shared between the eSPI master and eSPI slaves.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Adddress of the representation of read flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

int `espi_write_flash`(const struct *device* *dev, struct *espi_flash_packet* *pckt)

Sends a write request packet for shared flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI master and eSPI slaves.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the representation of write flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

int `espi_flash_erase`(const struct *device* *dev, struct *espi_flash_packet* *pckt)

Sends a write request packet for shared flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI master and eSPI slaves.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the representation of write flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

static inline void `espi_init_callback`(struct espi_callback *callback, *espi_callback_handler_t* handler, enum *espi_bus_event* evt_type)

Helper to initialize a struct espi_callback properly.

Callback model

+—-—+ +———-—+ +——+ +———+ | App | | eSPI driver | | HW | |eSPI Host| +—+—+ +—-—+–—+ +—+—+ +-—+-—+ | | | | |

espi_init_callback | | | +————————————-—> | | | | espi_add_callback | | +—————————————>+ | | | | eSPI reset | eSPI host | | IRQ +<—————+ resets the

	<—————+	bus
	Processed	
	within the	
	driver	

| | | VW CH ready| eSPI host | | IRQ +<—————+ enables VW

	<—————+	channel
	Processed	
	within the	
	driver	

| | | Memory I/O | Peripheral | | <————-—+ event | +<—————+ | +<—————————————+ callback | | | Report peripheral event | | | | and data for the event | | | | | | | | | SLP_S5 | eSPI host | | <————-—+ send VWire | +<—————+ | +<—————————————+ callback | | | App enables/configures | | |

discrete regulator		
espi_send_vwire_signal		

+—————————————>—————>|—————>| | | | | | | | HOST_RST | eSPI host | | <————-—+ send VWire | +<—————+ | +<—————————————+ callback | | | App reset host-related | | | | data structures | | | | | | | | | | C10 | eSPI host | | +<—————+ send VWire | <————-—+ | <—————————————+ | | | App executes | | |

- power mgmt policy | | |

Parameters

- `callback` – A valid Application's callback structure pointer.
- `handler` – A valid handler function pointer.
- `evt_type` – indicates the eSPI event relevant for the handler. for VWIRE_RECEIVED event the data will indicate the new level asserted

static inline int `espi_add_callback`(const struct *device* *dev, struct espi_callback *callback)

Add an application callback.

Note: enables to add as many callback as needed on the same device.

Note: Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current eSPI event is not specified.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – A valid Application's callback structure pointer.

Returns 0 if successful, negative errno code on failure.

static inline int `espi_remove_callback`(const struct *device* *dev, struct espi_callback *callback)

Remove an application callback.

Note: enables to remove as many callbacks as added through *espi_add_callback()*.

Warning: It is explicitly permitted, within a callback handler, to remove the registration for the callback that is running, i.e. `callback`. Attempts to remove other registrations on the same device may result in undefined behavior, including failure to invoke callbacks that remain registered and unintended invocation of removed callbacks.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – A valid application's callback structure pointer.

Returns 0 if successful, negative errno code on failure.

int `espi_saf_config`(const struct *device* *dev, const struct *espi_saf_cfg* *cfg)

Configure operation of a eSPI controller.

This routine provides a generic interface to override eSPI controller capabilities.

If this eSPI controller is acting as slave, the values set here will be discovered as part through the GET_CONFIGURATION command issued by the eSPI master during initialization.

If this eSPI controller is acting as master, the values set here will be used by eSPI master to determine minimum common capabilities with eSPI slave then send via SET_CONFIGURATION command.

+——+ +———+ +——+ +———+ +———+ | eSPI | | eSPI | | eSPI | | eSPI | | eSPI | | slave | | driver | | bus | | driver | | host | +——+ +———+ +——+ +———+ +———+ | | | | | | espi_config | Set eSPI | Set eSPI | espi_config | +—————+ ctrl regs | cap ctrl reg| +———+ | +—-—+ | +——+ |

	<——+	+—-—>	
		GET_CONFIGURATION	

| | +<———————+ | | |<————| | | | | eSPI caps | | | | |————>+ response | | | | |———————>+ | | | | | | | | SET_CONFIGURATION | | | | +<———————+ | | | | accept | | | | +———————>+ |

- + + + +

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `cfg` – the device runtime configuration for the eSPI controller.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.
- `-EINVAL` – invalid capabilities, failed to configure device.
- `-ENOTSUP` – capability not supported by eSPI slave.

int `espi_saf_set_protection_regions`(const struct *device* *dev, const struct espi_saf_protection *pr)

Set one or more SAF protection regions.

This routine provides an interface to override the default flash protection regions of the SAF controller.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pr` – Pointer to the SAF protection region structure.

Return values

- `0` – If successful.
- `-EIO` – General input / output error, failed to configure device.
- `-EINVAL` – invalid capabilities, failed to configure device.
- `-ENOTSUP` – capability not supported by eSPI slave.

int `espi_saf_activate`(const struct *device* *dev)

Activate SAF block.

This routine activates the SAF block and should only be called after SAF has been configured and the eSPI Master has enabled the Flash Channel.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `0` – If successful
- `-EINVAL` – if failed to activate SAF.

bool `espi_saf_get_channel_status`(const struct *device* *dev)

Query to see if SAF is ready.

This routine allows to check if SAF is ready before use.

Parameters

- `dev` – Pointer to the device structure for the driver instance.

Return values

- `true` – If eSPI SAF is ready.
- `false` – otherwise.

int `espi_saf_flash_read`(const struct *device* *dev, struct *espi_saf_packet* *pckt)

Sends a read request packet for slave attached flash.

This routines provides an interface to send a request to read the flash component shared between the eSPI master and eSPI slaves.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Adddress of the representation of read flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

int `espi_saf_flash_write`(const struct *device* *dev, struct *espi_saf_packet* *pckt)

Sends a write request packet for slave attached flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI master and eSPI slaves.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the representation of write flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

int `espi_saf_flash_erase`(const struct *device* *dev, struct *espi_saf_packet* *pckt)

Sends a write request packet for slave attached flash.

This routines provides an interface to send a request to write to the flash components shared between the eSPI master and eSPI slaves.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `pckt` – Address of the representation of erase flash transaction.

Return values

- `-ENOTSUP` – eSPI flash logical channel transactions not supported.
- `-EBUSY` – eSPI flash channel is not ready or disabled by master.
- `-EIO` – General input / output error, failed request to master.

static inline void `espi_saf_init_callback`(struct espi_callback *callback,
espi_callback_handler_t handler, enum *espi_bus_event* evt_type)

Helper to initialize a struct espi_callback properly.

Callback model

+—-—+ +———-—+ +——+ +———+ | App | | eSPI driver | | HW | |eSPI Host| +—+—+ +—-—+–—+ +—+—+ +-—+-—+ | | | | | espi_init_callback | | | +——————-—> | | | | espi_add_callback | | +———————>+ | | | | eSPI reset | eSPI host | | IRQ +<————+ resets the

		<————+	bus
	Processed	
	within the	
	driver	

| | | VW CH ready| eSPI host | | IRQ +<————+ enables VW

		<————+	channel
	Processed	
	within the	
	driver	

| | | Memory I/O | Peripheral | | <———-—+ event | +<————+ | +<———————+ callback | | | Report peripheral event | | | | and data for the event | | | | | | | | | SLP_S5 | eSPI host | | <———-—+ send VWire | +<————+ | +<———————+ callback | | | App enables/configures | | |

discrete regulator		
espi_send_vwire_signal		

+———————>————>|————>| | | | | | | | HOST_RST | eSPI host | | <———-—+ send VWire | +<————+ | +<———————+ callback | | | App reset host-related | | | | data structures | | | | | | | | | C10 | eSPI host | | +<————+ send VWire | <———-—+ | <———————+ | | | App executes | | |

- power mgmt policy | | |

Parameters

- `callback` – A valid Application's callback structure pointer.
- `handler` – A valid handler function pointer.
- `evt_type` – indicates the eSPI event relevant for the handler. for VWIRE_RECEIVED event the data will indicate the new level asserted

static inline int `espi_saf_add_callback`(const struct *device* *dev, struct espi_callback *callback)

Add an application callback.

Note: enables to add as many callback as needed on the same device.

Note: Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current eSPI event is not specified.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – A valid Application's callback structure pointer.

Returns 0 if successful, negative errno code on failure.

static inline int `espi_saf_remove_callback`(const struct *device* *dev, struct espi_callback *callback)

Remove an application callback.

Note: enables to remove as many callbacks as added through *espi_add_callback()*.

Warning: It is explicitly permitted, within a callback handler, to remove the registration for the callback that is running, i.e. `callback`. Attempts to remove other registrations on the same device may result in undefined behavior, including failure to invoke callbacks that remain registered and unintended invocation of removed callbacks.

Parameters

- `dev` – Pointer to the device structure for the driver instance.
- `callback` – A valid application's callback structure pointer.

Returns 0 if successful, negative errno code on failure.

struct `espi_evt_data_kbc`

#include <espi.h> Bit field definition of evt_data in struct *espi_event* for KBC.

struct `espi_evt_data_acpi`

#include <espi.h> Bit field definition of evt_data in struct *espi_event* for ACPI.

struct `espi_event`

#include <espi.h> eSPI event

Public Members

enum *espi_bus_event* `evt_type`
Event type

uint32_t `evt_details`
Additional details for bus event type

uint32_t `evt_data`
Data associated to the event

struct `espi_cfg`
#include <espi.h> eSPI bus configuration parameters

Public Members

enum *espi_io_mode* `io_caps`
Supported I/O mode

enum *espi_channel* `channel_caps`
Supported channels

uint8_t `max_freq`
Maximum supported frequency in MHz

struct `espi_request_packet`
#include <espi.h> eSPI peripheral request packet format

struct `espi_oob_packet`
#include <espi.h> eSPI out-of-band transaction packet format

struct `espi_flash_packet`
#include <espi.h> eSPI flash transactions packet format

struct `espi_saf_cfg`
#include <espi_saf.h> eSPI SAF configuration parameters

struct `espi_saf_packet`
#include <espi_saf.h> eSPI SAF transaction packet format

7.22 Power Management

Zephyr RTOS power management subsystem provides several means for a system integrator to implement power management support that can take full advantage of the power saving features of SOCs.

7.22.1 Terminology

SOC interface This is a general term for the components that have knowledge of the SOC and provide interfaces to the hardware features. It will abstract the SOC specific implementations to the applications and the OS.

Idle Thread A system thread that runs when there are no other threads ready to run.

Power gating Power gating reduces power consumption by shutting off current to blocks of the integrated circuit that are not in use.

Power State SOC Power State describes processor and device power states implemented at the SOC level. Power states are represented by `pm_state` and each one has a different meaning.

Device Runtime Power Management Device Runtime Power Management (PM) refers the capability of devices be able of saving energy independently of the the system. Devices will keep reference of their usage and will automatically be suspended or resumed. This feature is enabled via the :`CONFIG_PM_DEVICE_RUNTIME` Kconfig option.

7.22.2 Overview

The interfaces and APIs provided by the power management subsystem are designed to be architecture and SOC independent. This enables power management implementations to be easily adapted to different SOCs and architectures.

The architecture and SOC independence is achieved by separating the core infrastructure and the SOC specific implementations. The SOC specific implementations are abstracted to the application and the OS using hardware abstraction layers.

The power management features are classified into the following categories.

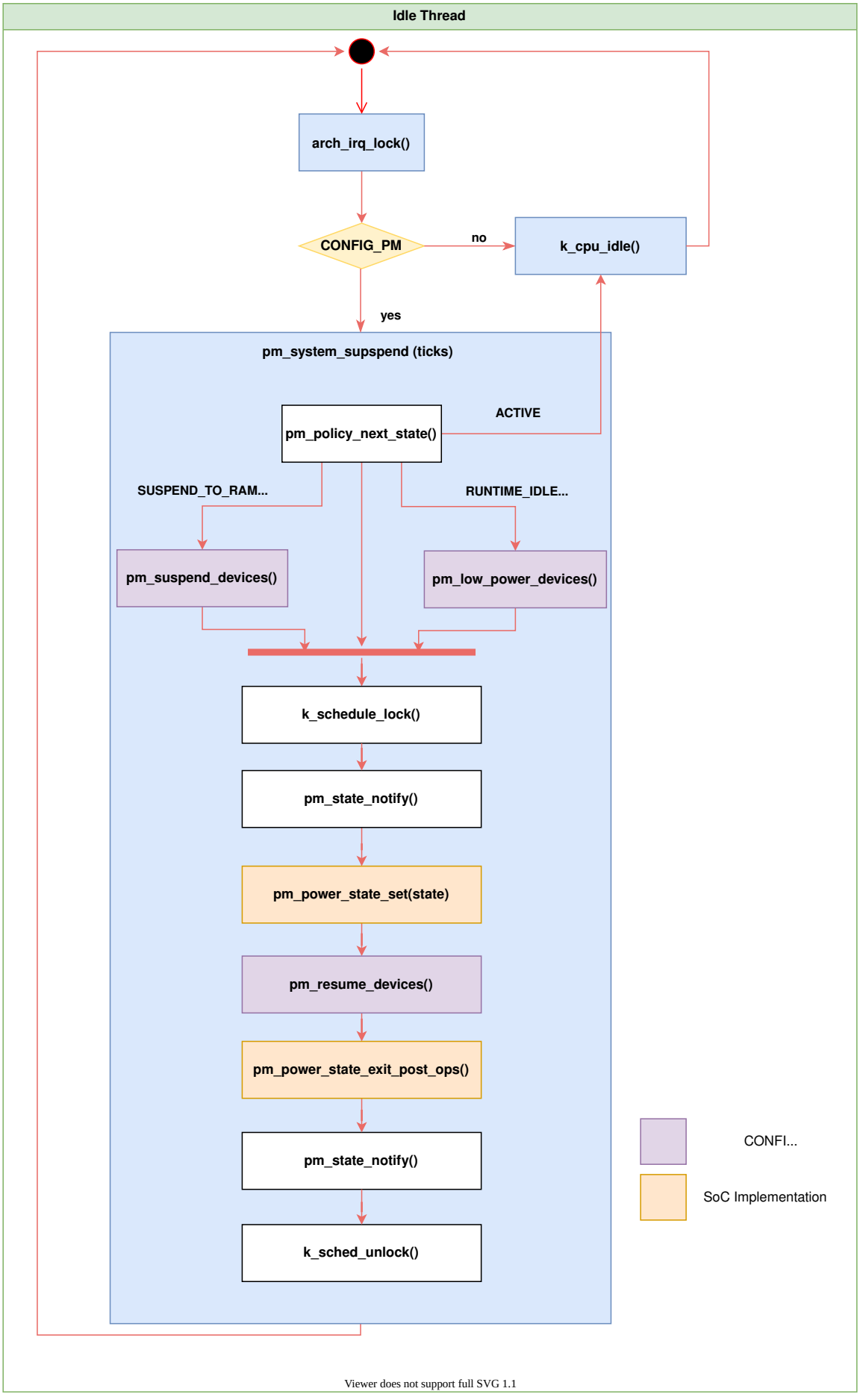
- System Power Management
- Device Power Management

7.22.3 System Power Management

The kernel enters the idle state when it has nothing to schedule. If enabled via the `CONFIG_PM` Kconfig option, the Power Management Subsystem can put an idle system in one of the supported power states, based on the selected power management policy and the duration of the idle time allotted by the kernel.

It is an application responsibility to set up a wake up event. A wake up event will typically be an interrupt triggered by one of the SoC peripheral modules such as a SysTick, RTC, counter, or GPIO. Depending on the power mode entered, only some SoC peripheral modules may be active and can be used as a wake up source.

The following diagram describes system power management:


Idle Thread
arch_irq_lock()
CONFIG_PM
no
k_cpu_idle()
yes
pm_system_supspend (ticks)
pm_policy_next_state()
ACTIVE
SUSPEND_TO_RAM...
RUNTIME_IDLE...
pm_suspend_devices()
pm_low_power_devices()
k_schedule_lock()
pm_state_notify()
pm_power_state_set(state)
pm_resume_devices()
pm_power_state_exit_post_ops()
pm_state_notify()
k_sched_unlock()
CONFI...
SoC Implementation
Viewer does not support full SVG 1.1

Some handful examples using different power management features:

- samples/boards/stm32/power_mgmt/blinky/
- samples/boards/nrf/system_off/
- samples/subsys/pm/device_pm/
- tests/subsys/pm/power_mgmt/
- tests/subsys/pm/power_mgmt_soc/
- tests/subsys/pm/power_state_api/

Power States

The power management subsystem contains a set of states based on power consumption and context retention.

The list of available power states is defined by `pm_state`. In general power states with higher indexes will offer greater power savings and have higher wake latencies. Following is a thorough list of available states:

enumerator `PM_STATE_ACTIVE`

Runtime active state.

The system is fully powered and active.

Note: This state is correlated with ACPI G0/S0 state

enumerator `PM_STATE_RUNTIME_IDLE`

Runtime idle state.

Runtime idle is a system sleep state in which all of the cores enter deepest possible idle state and wait for interrupts, no requirements for the devices, leaving them at the states where they are.

Note: This state is correlated with ACPI S0ix state

enumerator `PM_STATE_SUSPEND_TO_IDLE`

Suspend to idle state.

The system goes through a normal platform suspend where it puts all of the cores in deepest possible idle state and *may* puts peripherals into low-power states. No operating state is lost (ie. the cpu core does not lose execution context), so the system can go back to where it left off easily enough.

Note: This state is correlated with ACPI S1 state

enumerator `PM_STATE_STANDBY`

Standby state.

In addition to putting peripherals into low-power states all non-boot CPUs are powered off. It should allow more energy to be saved relative to suspend to idle, but the resume latency will generally be greater than for that state. But it should be the same state with suspend to idle state on uniprocesser system.

Note: This state is correlated with ACPI S2 state

enumerator PM_STATE_SUSPEND_TO_RAM

Suspend to ram state.

This state offers significant energy savings by powering off as much of the system as possible, where memory should be placed into the self-refresh mode to retain its contents. The state of devices and CPUs is saved and held in memory, and it may require some boot- strapping code in ROM to resume the system from it.

Note: This state is correlated with ACPI S3 state

enumerator PM_STATE_SUSPEND_TO_DISK

Suspend to disk state.

This state offers significant energy savings by powering off as much of the system as possible, including the memory. The contents of memory are written to disk or other non-volatile storage, and on resume it's read back into memory with the help of boot-strapping code, restores the system to the same point of execution where it went to suspend to disk.

Note: This state is correlated with ACPI S4 state

enumerator PM_STATE_SOFT_OFF

Soft off state.

This state consumes a minimal amount of power and requires a large latency in order to return to runtime active state. The contents of system(CPU and memory) will not be preserved, so the system will be restarted as if from initial power-up and kernel boot.

Note: This state is correlated with ACPI G2/S5 state

Power States Constraint

The power management subsystem allows different Zephyr components and applications to set constraints on various power states preventing the system from transitiioning into these states. This can be used by devices when executing tasks in background to avoid the system to go to a specific state where it would lose context. Constraints can be set, released and checked using the follow APIs:

void pm_constraint_set(enum pm_state state)

Set a constraint for a power state.

Disabled state cannot be selected by the Zephyr power management policies. Application defined policy should use the *pm_constraint_get* function to check if given state is enabled and could be used.

Note: This API is refcount

Parameters

- state – **[in]** Power state to be disabled.

void `pm_constraint_release`(enum pm_state state)

Release a constraint for a power state.

Enabled state can be selected by the Zephyr power management policies. Application defined policy should use the *pm_constraint_get* function to check if given state is enabled and could be used. By default all power states are enabled.

Note: This API is refcount

Parameters

- `state` – **[in]** Power state to be enabled.

bool `pm_constraint_get`(enum pm_state state)

Check if particular power state is enabled.

This function returns true if given power state is enabled.

Parameters

- `state` – **[in]** Power state.

Power Management Policies

The power management subsystem supports the following power management policies:

- Residency based
- Application defined

The policy manager is responsible for informing the power subsystem which power state the system should transition to based on states defined by the platform and possible runtime *constraints*

Information about states can be found in the device tree, see dts/bindings/power/state.yaml.

Residency The power management system enters the power state which offers the highest power savings, and with a minimum residency value (in device tree, see dts/bindings/power/state.yaml) less than or equal to the scheduled system idle time duration.

This policy also accounts for the time necessary to become active again. The core logic used by this policy to select the best power state is:

```
if (time_to_next_scheduled_event >= (state.min_residency_us + state.exit_latency))) {
   return state
}
```

Application The power management policy is defined by the application which has to implement the following function.

```
struct pm_state_info pm_policy_next_state(int32_t ticks);
```

In this policy the application is free to decide which power state the system should transition to based on the remaining time for the next scheduled timeout.

An example of an application that defines its own policy can be found in tests/subsys/pm/power_mgmt/.

7.22.4 Device Power Management Infrastructure

The device power management infrastructure consists of interfaces to the *Device Driver Model*. These APIs send control commands to the device driver to update its power state or to get its current power state.

Zephyr RTOS supports two methods of doing device power management.

- Runtime Device Power Management
- System Power Management

Runtime Device Power Management

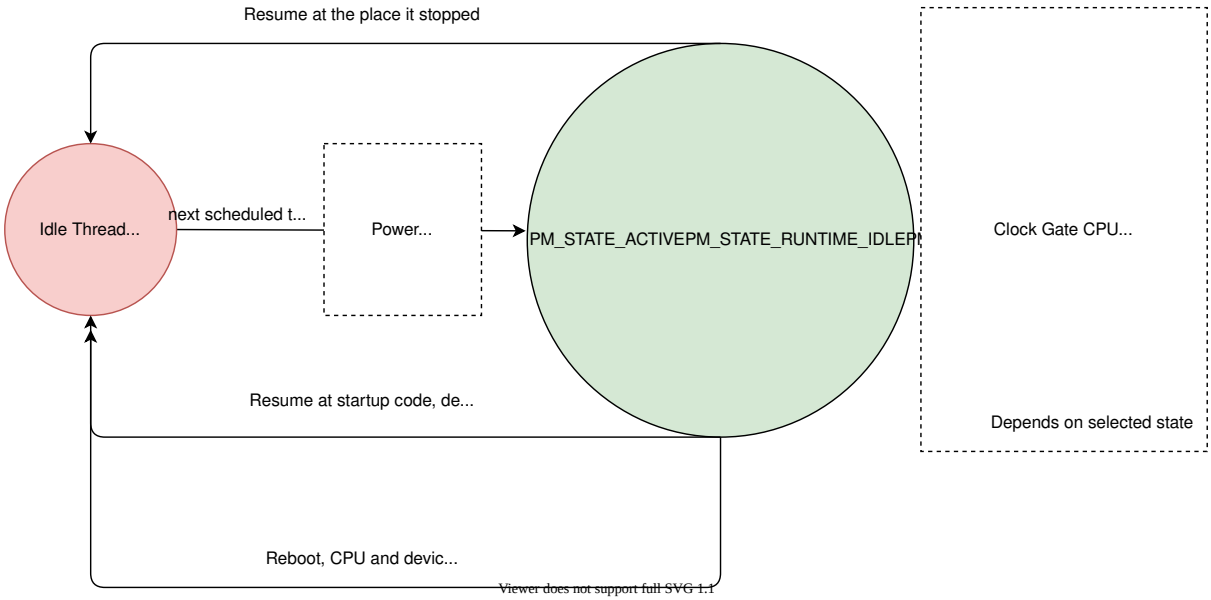
In this method, the application or any component that deals with devices directly and has the best knowledge of their use, performs the device power management. This saves power if some devices that are not in use can be turned off or put in power saving mode. This method allows saving power even when the CPU is active. The components that use the devices need to be power aware and should be able to make decisions related to managing device power.

In this method, the SOC interface can enter CPU or SOC power states quickly when `pm_system_suspend()` gets called. This is because it does not need to spend time doing device power management if the devices are already put in the appropriate power state by the application or component managing the devices.

System Power Management

In this method device power management is mostly done inside `pm_system_suspend()` along with entering a CPU or SOC power state.

If a decision to enter a lower power state is made, the implementation would enter it only after checking if the devices are not in the middle of a hardware transaction that cannot be interrupted. This method can be used in implementations where the applications and components using devices are not expected to be power aware and do not implement runtime device power management.



This method can also be used to emulate a hardware feature supported by some SOCs which triggers automatic entry to a lower power state when all devices are idle. Refer to *Busy Status Indication* to see how to indicate whether a device is busy or idle.

Device Power Management States

The power management subsystem defines four device states. These states are classified based on the degree of device context that gets lost in those states, kind of operations done to save power, and the impact on the device behavior due to the state transition. Device context includes device registers, clocks, memory etc.

The four device power states:

`PM_DEVICE_STATE_ACTIVE`

Normal operation of the device. All device context is retained.

`PM_DEVICE_STATE_LOW_POWER`

Device context is preserved by the HW and need not be restored by the driver.

`PM_DEVICE_STATE_SUSPENDED`

Most device context is lost by the hardware. Device drivers must save and restore or reinitialize any context lost by the hardware.

`PM_DEVICE_STATE_OFF`

Power has been fully removed from the device. The device context is lost when this state is entered. Need to reinitialize the device when powering it back on.

Device Power Management Operations

Zephyr RTOS power management subsystem provides a control function interface to device drivers to indicate power management operations to perform. Each device driver defines:

- The device's supported power states.
- The device's supported transitions between power states.
- The device's necessary operations to handle the transition between power states.

The following are some examples of operations that the device driver may perform in transition between power states:

- Save/Restore device states.
- Gate/Un-gate clocks.
- Gate/Un-gate power.
- Mask/Un-mask interrupts.

Device Model with Power Management Support

Drivers initialize the devices using macros. See *Device Driver Model* for details on how these macros are used. Use the DEVICE_DEFINE macro to initialize drivers providing power management support via the PM control function. One of the macro parameters is the pointer to the pm_control handler function. If the driver doesn't implement any power control operations, it can initialize the corresponding pointer with `NULL`.

Device Power Management API

The SOC interface and application use these APIs to perform power management operations on the devices.

Get Device List

```
size_t z_device_get_all_static(struct device const **device_list);
```

The Zephyr RTOS kernel internally maintains a list of all devices in the system. The SOC interface uses this API to get the device list. The SOC interface can use the list to identify the devices on which to execute power management operations.

Note: Ensure that the SOC interface does not alter the original list. Since the kernel uses the original list, it must remain unchanged.

Device Set Power State

```
int pm_device_state_set(const struct device *dev, enum pm_device_state state);
```

Calls the `pm_control()` handler function implemented by the device driver with the provided state.

Device Get Power State

```
int pm_device_state_get(const struct device *dev, enum pm_device_state *state);
```

Busy Status Indication

The SOC interface executes some power policies that can turn off power to devices, causing them to lose their state. If the devices are in the middle of some hardware transaction, like writing to flash memory when the power is turned off, then such transactions would be left in an inconsistent state. This infrastructure guards such transactions by indicating to the SOC interface that the device is in the middle of a hardware transaction.

When the `pm_system_suspend()` is called, depending on the power state returned by the policy manager, the system may suspend or put devices in low power if they are not marked as busy.

Here are the APIs used to set, clear, and check the busy status of devices.

Indicate Busy Status API

```
void device_busy_set(const struct device *busy_dev);
```

Sets a bit corresponding to the device, in a data structure maintained by the kernel, to indicate whether or not it is in the middle of a transaction.

Clear Busy Status API

```
void device_busy_clear(const struct device *busy_dev);
```

Clears the bit corresponding to the device in a data structure maintained by the kernel to indicate that the device is not in the middle of a transaction.

Check Busy Status of Single Device API

```
int device_busy_check(const struct device *chk_dev);
```

Checks whether a device is busy. The API returns 0 if the device is not busy.

This API is used by the system power management.

Check Busy Status of All Devices API

```
int device_any_busy_check(void);
```

Checks if any device is busy. The API returns 0 if no device in the system is busy.

Wakeup capability Some devices are capable of waking the system up from a sleep state. When a device has such capability, applications can enable or disable this feature on a device dynamically using *pm_device_wakeup_enable()*.

This property can be set on device declaring the property `wakeup-source` in the device node in devicetree. For example, this devicetree fragment sets the `gpio0` device as a "wakeup" source:

```
gpio0: gpio@40022000 {
        compatible = "ti,cc13xx-cc26xx-gpio";
        reg = <0x40022000 0x400>;
        interrupts = <0 0>;
        status = "disabled";
        label = "GPIO_0";
        gpio-controller;
        wakeup-source;
        #gpio-cells = <2>;
};
```

By default, "wakeup" capable devices do not have this functionality enabled during the device initialization. Applications can enable this functionality later calling *pm_device_wakeup_enable()*.

Note: This property is **only** used by the system power management to identify devices that should not be suspended. It is responsability of driver or the application to do any additional configuration required by the device to support it.

7.22.5 Device Runtime Power Management

The Device Runtime Power Management framework is an Active Power Management mechanism which reduces the overall system Power consumtion by suspending the devices which are idle or not being used while the System is active or running.

The framework uses *pm_device_state_set()* API set the device power state accordingly based on the usage count.

The interfaces and APIs provided by the Device Runtime PM are designed to be generic and architecture independent.

Device Runtime Power Management API

The Device Drivers use these APIs to perform device runtime power management operations on the devices.

Enable Device Runtime Power Management of a Device API

```
void pm_device_enable(const struct device *dev);
```

Enables Runtime Power Management of the device.

Disable Device Runtime Power Management of a Device API

```
void pm_device_disable(const struct device *dev);
```

Disables Runtime Power Management of the device.

Resume Device asynchronously API

```
int pm_device_get_async(const struct device *dev);
```

Marks the device as being used. This API will asynchronously bring the device to resume state if it was suspended. If the device was already active, it just increments the device usage count. The API returns 0 on success.

Device drivers can monitor this operation to finish calling `pm_device_wait()`.

Resume Device synchronously API

```
int pm_device_get(const struct device *dev);
```

Marks the device as being used. It will bring up or resume the device if it is in suspended state based on the device usage count. This call is blocked until the device PM state is changed to active. The API returns 0 on success.

Suspend Device asynchronously API

```
int pm_device_put_async(const struct device *dev);
```

Releases a device. This API asynchronously puts the device to suspend state if not already in suspend state if the usage count of this device reaches 0.

Device drivers can monitor this operation to finish calling `pm_device_wait()`.

Suspend Device synchronously API

```
int pm_device_put(const struct device *dev);
```

Marks the device as being released. It will put the device to suspended state if is is in active state based on the device usage count. This call is blocked until the device PM state is changed to resume. The API returns 0 on success. This call is blocked until the device is suspended.

7.22.6 Power Management Configuration Flags

The Power Management features can be individually enabled and disabled using the following configuration flags.

`CONFIG_PM`

This flag enables the power management subsystem.

`CONFIG_PM_DEVICE`

This flag is enabled if the SOC interface and the devices support device power management.

`CONFIG_PM_DEVICE_RUNTIME`

This flag enables the Runtime Power Management.

7.22.7 API Reference

Power Management Hook Interface

group `power_management_hook_interface`

Power Management Hooks.

Functions

void `pm_power_state_set`(struct pm_state_info info)

Put processor into a power state.

This function implements the SoC specific details necessary to put the processor into available power states.

Parameters

- `info` – Power state which should be used in the ongoing suspend operation.

void `pm_power_state_exit_post_ops`(struct pm_state_info info)

Do any SoC or architecture specific post ops after sleep state exits.

This function is a place holder to do any operations that may be needed to be done after sleep state exits. Currently it enables interrupts after resuming from sleep state. In future, the enabling of interrupts may be moved into the kernel.

System Power Management APIs

group `system_power_management_api`

System Power Management API.

Functions

void `pm_power_state_force`(struct pm_state_info info)

Force usage of given power state.

This function overrides decision made by PM policy forcing usage of given power state immediately.

Note: This function can only run in thread context

Parameters

- `info` – Power state which should be used in the ongoing suspend operation.

static inline void `pm_dump_debug_info`(void)

void `pm_notifier_register`(struct *pm_notifier* *notifier)

Register a power management notifier.

Register the given notifier from the power management notification list.

Parameters

- `notifier` – *pm_notifier* object to be registered.

int pm_notifier_unregister(struct *pm_notifier* *notifier)

Unregister a power management notifier.

Remove the given notifier from the power management notification list. After that this object callbacks will not be called.

Parameters

- notifier – *pm_notifier* object to be unregistered.

Returns 0 if the notifier was successfully removed, a negative value otherwise.

struct pm_notifier

#include <pm.h> Power management notifier struct

This struct contains callbacks that are called when the target enters and exits power states.

As currently implemented the entry callback is invoked when transitioning from PM_STATE_ACTIVE to another state, and the exit callback is invoked when transitioning from a non-active state to PM_STATE_ACTIVE. This behavior may change in the future.

Note: These callbacks can be called from the ISR of the event that caused the kernel exit from idling.

Note: It is not allowed to call *pm_notifier_unregister* or *pm_notifier_register* from these callbacks because they are called with the spin locked in those functions.

Public Members

void (*state_entry)(enum pm_state state)

Application defined function for doing any target specific operations for power state entry.

void (*state_exit)(enum pm_state state)

Application defined function for doing any target specific operations for power state exit.

Device Power Management APIs

group device_power_management_api

Device Power Management API.

Defines

INIT_PM_DEVICE_RUNTIME(obj)

device_pm_control_nop

Alias for legacy use of device_pm_control_nop

Typedefs

typedef int (*pm_device_control_callback_t)(const struct *device* *dev, enum *pm_device_action* action)

Device power management control function callback.

Param dev Device instance.

Param action Requested action.

Retval 0 If successful.

Retval -ENOTSUP If the requested action is not supported.

Retval Errno Other negative errno on failure.

Enums

enum pm_device_state

Device power states.

Values:

enumerator PM_DEVICE_STATE_ACTIVE

Device is in active or regular state.

enumerator PM_DEVICE_STATE_LOW_POWER

Device is in low power state.

Note: Device context is preserved.

---

enumerator PM_DEVICE_STATE_SUSPENDED

Device is suspended.

Note: Device context may be lost.

---

enumerator PM_DEVICE_STATE_OFF

Device is turned off (power removed).

Note: Device context is lost.

---

enum pm_device_flag

Device PM flags.

Values:

enumerator PM_DEVICE_FLAG_BUSY

Indicate if the device is busy or not.

enumerator PM_DEVICE_FLAGS_WS_CAPABLE

Indicates whether or not the device is capable of waking the system up.

enumerator PM_DEVICE_FLAGS_WS_ENABLED

Indicates if the device is being used as wakeup source.

enumerator PM_DEVICE_FLAG_TRANSITIONING

Indicates that the device is changing its state

enumerator PM_DEVICE_FLAG_COUNT

Number of flags (internal use only).

enum pm_device_action

Device PM actions.

Values:

enumerator PM_DEVICE_ACTION_SUSPEND

Suspend.

enumerator PM_DEVICE_ACTION_RESUME

Resume.

enumerator PM_DEVICE_ACTION_TURN_OFF

Turn off.

enumerator PM_DEVICE_ACTION_FORCE_SUSPEND

Force suspend.

enumerator PM_DEVICE_ACTION_LOW_POWER

Low power.

Functions

const char *pm_device_state_str(enum *pm_device_state* state)

Get name of device PM state.

Parameters

- state – State id which name should be returned

int pm_device_state_set(const struct *device* *dev, enum *pm_device_state* state)

Set the power state of a device.

This function calls the device PM control callback so that the device does the necessary operations to put the device into the given state.

Note: Some devices may not support all device power states.

Parameters

- dev – Device instance.

- `state` – Device power state to be set.

Return values

- `0` – If successful.
- `-ENOTSUP` – If requested state is not supported.
- `-EALREADY` – If device is already at the requested state.
- `-EBUSY` – If device is changing its state.
- `Errno` – Other negative errno on failure.

int `pm_device_state_get`(const struct *device* *dev, enum *pm_device_state* *state)

Obtain the power state of a device.

Parameters

- `dev` – Device instance.
- `state` – Pointer where device power state will be stored.

Return values

- `0` – If successful.
- `-ENOSYS` – If device does not implement power management.

void `pm_device_busy_set`(const struct *device* *dev)

Indicate that the device is in the middle of a transaction.

Called by a device driver to indicate that it is in the middle of a transaction.

Parameters

- `dev` – Pointer to device structure of the driver instance.

void `pm_device_busy_clear`(const struct *device* *dev)

Indicate that the device has completed its transaction.

Called by a device driver to indicate the end of a transaction.

Parameters

- `dev` – Pointer to device structure of the driver instance.

bool `pm_device_is_any_busy`(void)

Check if any device is in the middle of a transaction.

Called by an application to see if any device is in the middle of a critical transaction that cannot be interrupted.

Return values

- `false` – if no device is busy
- `true` – if any device is busy

bool `pm_device_is_busy`(const struct *device* *dev)

Check if a specific device is in the middle of a transaction.

Called by an application to see if a particular device is in the middle of a critical transaction that cannot be interrupted.

Parameters

- `dev` – Pointer to device structure of the specific device driver the caller is interested in.

Return values

- `false` – if the device is not busy
- `true` – if the device is busy

static inline void `device_busy_set`(const struct *device* *dev)

static inline void `device_busy_clear`(const struct *device* *dev)

static inline int `device_any_busy_check`(void)

static inline int `device_busy_check`(const struct *device* *dev)

bool `pm_device_wakeup_enable`(struct *device* *dev, bool enable)

Enable a power management wakeup source.

Enable a wakeup source. This will keep the current device active when the system is suspended, allowing it to be used to wake up the system.

Parameters

- `dev` – device object to enable.
- `enable` – `true` to enable or `false` to disable

Return values

- `true` – if the wakeup source was successfully enabled.
- `false` – if the wakeup source was not successfully enabled.

bool `pm_device_wakeup_is_enabled`(const struct *device* *dev)

Check if a power management wakeup source is enabled.

Checks if a wake up source is enabled.

Parameters

- `dev` – device object to check.

Return values

- `true` – if the wakeup source is enabled.
- `false` – if the wakeup source is not enabled.

bool `pm_device_wakeup_is_capable`(const struct *device* *dev)

Check if a device is wake up capable.

Parameters

- `dev` – device object to check.

Return values

- `true` – if the device is wake up capable.
- `false` – if the device is not wake up capable.

struct `pm_device`

#include <device.h> Device PM info.

Public Members

const struct *device* *`dev`

Pointer to the device

struct *k_mutex* `lock`
: Lock to synchronize the get/put operations

bool `enable`
: Device pm enable flag

uint32_t `usage`
: Device usage count

struct *k_work_delayable* `work`
: Work object for asynchronous calls

struct k_condvar `condvar`
: Event conditional var to listen to the sync request events

enum *pm_device_state* `state`
: Device power state

7.23 Random Number Generation

The random API subsystem provides random number generation APIs in both cryptographically and non-cryptographically secure instances. Which random API to use is based on the cryptographic requirements of the random number. The non-cryptographic APIs will return random values much faster if non-cryptographic values are needed.

The cryptographically secure random functions shall be compliant to the FIPS 140-2 [**?**] recommended algorithms. Hardware based random-number generators (RNG) can be used on platforms with appropriate hardware support. Platforms without hardware RNG support shall use the CTR-DRBG algorithm. The algorithm can be provided by TinyCrypt or mbedTLS depending on your application performance and resource requirements.

Note: The CTR-DRBG generator needs an entropy source to establish and maintain the cryptographic security of the PRNG.

7.23.1 Kconfig Options

These options can be found in the following path subsys/random/Kconfig.

`CONFIG_TEST_RANDOM_GENERATOR` For testing, this option permits random number APIs to return values that are not truly random.

The random number generator choice group allows selection of the RNG source function for the system via the RNG_GENERATOR_CHOICE choice group. An override of the default value can be specified in the SOC or board .defconfig file by using:

```
choice RNG_GENERATOR_CHOICE
        default XOSHIRO_RANDOM_GENERATOR
endchoice
```

The random number generators available include:

`CONFIG_TIMER_RANDOM_GENERATOR` enables number generator based on system timer clock. This number generator is not random and used for testing only.

CONFIG_ENTROPY_DEVICE_RANDOM_GENERATOR enables a random number generator that uses the enabled hardware entropy gathering driver to generate random numbers.

CONFIG_XOSHIRO_RANDOM_GENERATOR enables the Xoshiro128++ pseudo-random number generator, that uses the entropy driver as a seed source.

The CSPRNG_GENERATOR_CHOICE choice group provides selection of the cryptographically secure random number generator source function. An override of the default value can be specified in the SOC or board .defconfig file by using:

```
choice CSPRNG_GENERATOR_CHOICE
        default CTR_DRBG_CSPRNG_GENERATOR
endchoice
```

The cryptographically secure random number generators available include:

CONFIG_HARDWARE_DEVICE_CS_GENERATOR enables a cryptographically secure random number generator using the hardware random generator driver

CONFIG_CTR_DRBG_CSPRNG_GENERATOR enables the CTR-DRBG pseudo-random number generator. The CTR-DRBG is a FIPS140-2 recommended cryptographically secure random number generator.

Personalization data can be provided in addition to the entropy source to make the initialization of the CTR-DRBG as unique as possible.

CONFIG_CS_CTR_DRBG_PERSONALIZATION CTR-DRBG Initialization Personalization string

7.23.2 API Reference

group random_api

Random Function APIs.

Functions

uint32_t sys_rand32_get(void)

Return a 32-bit random value that should pass general randomness tests.

Note: The random value returned is not a cryptographically secure random number value.

Returns 32-bit random value.

void sys_rand_get(void *dst, size_t len)

Fill the destination buffer with random data values that should pass general randomness tests.

Note: The random values returned are not considered cryptographically secure random number values.

Parameters

- dst – **[out]** destination buffer to fill with random data.
- len – size of the destination buffer.

int `sys_csrand_get`(void *dst, size_t len)

Fill the destination buffer with cryptographically secure random data values.

Note: If the random values requested do not need to be cryptographically secure then use *sys_rand_get()* instead.

Parameters

- `dst` – **[out]** destination buffer to fill.
- `len` – size of the destination buffer.

Returns 0 if success, -EIO if entropy reseed error

7.24 Resource Management

There are various situations where it's necessary to coordinate resource use at runtime among multiple clients. These include power rails, clocks, other peripherals, and binary device power management. The complexity of properly managing multiple consumers of a device in a multithreaded system, especially when transitions may be asynchronous, suggests that a shared implementation is desirable.

Zephyr provides managers for several coordination policies. These managers are embedded into services that use them for specific functions.

7.24.1 On-Off Manager

An on-off manager supports an arbitrary number of clients of a service which has a binary state. Example applications are power rails, clocks, and binary device power management.

The manager has the following properties:

- The stable states are off, on, and error. The service always begins in the off state. The service may also be in a transition to a given state.
- The core operations are request (add a dependency) and release (remove a dependency). Supporting operations are reset (to clear an error state) and cancel (to reclaim client data from an in-progress transition). The service manages the state based on calls to functions that initiate these operations.
- The service transitions from off to on when first client request is received.
- The service transitions from on to off when last client release is received.
- Each service configuration provides functions that implement the transition from off to on, from on to off, and optionally from an error state to off. Transitions must be invokable from both thread and interrupt context.
- The request and reset operations are asynchronous using *Asynchronous Notification APIs*. Both operations may be cancelled, but cancellation has no effect on the in-progress transition.
- Requests to turn on may be queued while a transition to off is in progress: when the service has turned off successfully it will be immediately turned on again (where context allows) and waiting clients notified when the start completes.

Requests are reference counted, but not tracked. That means clients are responsible for recording whether their requests were accepted, and for initiating a release only if they have previously successfully completed a request. Improper use of the API can cause an active client to be shut out, and the manager does not maintain a record of specific clients that have been granted a request.

Failures in executing a transition are recorded and inhibit further requests or releases until the manager is reset. Pending requests are notified (and cancelled) when errors are discovered.

Transition operation completion notifications are provided through *Asynchronous Notification APIs*.

Clients and other components interested in tracking all service state changes, including when a service begins turning off or enters an error state, can be informed of state transitions by registering a monitor with onoff_monitor_register(). Notification of changes are provided before issuing completion notifications associated with the new state.

Note: A generic API may be implemented by multiple drivers where the common case is asynchronous. The on-off client structure may be an appropriate solution for the generic API. Where drivers that can guarantee synchronous context-independent transitions a driver may use *onoff_sync_service* and its supporting API rather than *onoff_manager*, with only a small reduction in functionality (primarily no support for the monitor API).

---

group `resource_mgmt_onoff_apis`

Defines

`ONOFF_FLAG_ERROR`

Flag indicating an error state.

Error states are cleared using *onoff_reset()*.

`ONOFF_FLAG_ONOFF`

`ONOFF_FLAG_TRANSITION`

`ONOFF_STATE_MASK`

Mask used to isolate bits defining the service state.

Mask a value with this then test for ONOFF_FLAG_ERROR to determine whether the machine has an unfixed error, or compare against ONOFF_STATE_ON, ONOFF_STATE_OFF, ONOFF_STATE_TO_ON, ONOFF_STATE_TO_OFF, or ONOFF_STATE_RESETTING.

`ONOFF_STATE_OFF`

Value exposed by ONOFF_STATE_MASK when service is off.

`ONOFF_STATE_ON`

Value exposed by ONOFF_STATE_MASK when service is on.

`ONOFF_STATE_ERROR`

Value exposed by ONOFF_STATE_MASK when the service is in an error state (and not in the process of resetting its state).

`ONOFF_STATE_TO_ON`

Value exposed by ONOFF_STATE_MASK when service is transitioning to on.

ONOFF_STATE_TO_OFF

Value exposed by ONOFF_STATE_MASK when service is transitioning to off.

ONOFF_STATE_RESETTING

Value exposed by ONOFF_STATE_MASK when service is in the process of resetting.

ONOFF_TRANSITIONS_INITIALIZER(_start, _stop, _reset)

Initializer for a *onoff_transitions* object.

Parameters

- `_start` – a function used to transition from off to on state.
- `_stop` – a function used to transition from on to off state.
- `_reset` – a function used to clear errors and force the service to an off state. Can be null.

ONOFF_MANAGER_INITIALIZER(_transitions)

ONOFF_CLIENT_EXTENSION_POS

Identify region of *sys_notify* flags available for containing services.

Bits of the flags field of the *sys_notify* structure contained within the queued_operation structure at and above this position may be used by extensions to the *onoff_client* structure.

These bits are intended for use by containing service implementations to record client-specific information and are subject to other conditions of use specified on the *sys_notify* API.

Typedefs

typedef void (*onoff_notify_fn)(struct *onoff_manager* *mgr, int res)

Signature used to notify an on-off manager that a transition has completed.

Functions of this type are passed to service-specific transition functions to be used to report the completion of the operation. The functions may be invoked from any context.

Param mgr the manager for which transition was requested.

Param res the result of the transition. This shall be non-negative on success, or a negative error code. If an error is indicated the service shall enter an error state.

typedef void (*onoff_transition_fn)(struct *onoff_manager* *mgr, *onoff_notify_fn* notify)

Signature used by service implementations to effect a transition.

Service definitions use two required function pointers of this type to be notified that a transition is required, and a third optional one to reset the service when it is in an error state.

The start function will be called only from the off state.

The stop function will be called only from the on state.

The reset function (where supported) will be called only when *onoff_has_error()* returns true.

Note: All transitions functions must be isr-ok.

Param mgr the manager for which transition was requested.

Param notify the function to be invoked when the transition has completed. If the transition is synchronous, notify shall be invoked by the implementation before the transition function returns. Otherwise the implementation shall capture this parameter and invoke it when the transition completes.

typedef void (*`onoff_client_callback`)(struct *onoff_manager* *mgr, struct *onoff_client* *cli, uint32_t state, int res)

Signature used to notify an on-off service client of the completion of an operation.

These functions may be invoked from any context including pre-kernel, ISR, or cooperative or pre-emptible threads. Compatible functions must be isr-ok and not sleep.

Param mgr the manager for which the operation was initiated. This may be null if the on-off service uses synchronous transitions.

Param cli the client structure passed to the function that initiated the operation.

Param state the state of the machine at the time of completion, restricted by ONOFF_STATE_MASK. ONOFF_FLAG_ERROR must be checked independently of whether res is negative as a machine error may indicate that all future operations except *onoff_reset()* will fail.

Param res the result of the operation. Expected values are service-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed. If res is negative ONOFF_FLAG_ERROR will be set in state, but if res is non-negative ONOFF_FLAG_ERROR may still be set in state.

typedef void (*`onoff_monitor_callback`)(struct *onoff_manager* *mgr, struct *onoff_monitor* *mon, uint32_t state, int res)

Signature used to notify a monitor of an onoff service of errors or completion of a state transition.

This is similar to onoff_client_callback but provides information about all transitions, not just ones associated with a specific client. Monitor callbacks are invoked before any completion notifications associated with the state change are made.

These functions may be invoked from any context including pre-kernel, ISR, or cooperative or pre-emptible threads. Compatible functions must be isr-ok and not sleep.

The callback is permitted to unregister itself from the manager, but must not register or unregister any other monitors.

Param mgr the manager for which a transition has completed.

Param mon the monitor instance through which this notification arrived.

Param state the state of the machine at the time of completion, restricted by ONOFF_STATE_MASK. All valid states may be observed.

Param res the result of the operation. Expected values are service- and state-specific, but the value shall be non-negative if the operation succeeded, and negative if the operation failed.

Functions

int `onoff_manager_init`(struct *onoff_manager* *mgr, const struct *onoff_transitions* *transitions)

Initialize an on-off service to off state.

This function must be invoked exactly once per service instance, by the infrastructure that provides the service, and before any other on-off service API is invoked on the service.

This function should never be invoked by clients of an on-off service.

Parameters

- `mgr` – the manager definition object to be initialized.
- `transitions` – pointer to a structure providing transition functions. The referenced object must persist as long as the manager can be referenced.

Return values

- `0` – on success
- `-EINVAL` – if start, stop, or flags are invalid

static inline bool `onoff_has_error`(const struct *onoff_manager* *mgr)

Test whether an on-off service has recorded an error.

This function can be used to determine whether the service has recorded an error. Errors may be cleared by invoking *onoff_reset()*.

This is an unlocked convenience function suitable for use only when it is known that no other process might invoke an operation that transitions the service between an error and non-error state.

Returns true if and only if the service has an uncleared error.

int `onoff_request`(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Request a reservation to use an on-off service.

The return value indicates the success or failure of an attempt to initiate an operation to request the resource be made available. If initiation of the operation succeeds the result of the request operation is provided through the configured client notification method, possibly before this call returns.

Note that the call to this function may succeed in a case where the actual request fails. Always check the operation completion result.

Parameters

- `mgr` – the manager that will be used.
- `cli` – a non-null pointer to client state providing instructions on synchronous expectations and how to notify the client when the request completes. Behavior is undefined if client passes a pointer object associated with an incomplete service operation.

Return values

- `non-negative` – the observed state of the machine at the time the request was processed, if successful.
- `-EIO` – if service has recorded an an error.
- `-EINVAL` – if the parameters are invalid.
- `-EAGAIN` – if the reference count would overflow.

int `onoff_release`(struct *onoff_manager* *mgr)

Release a reserved use of an on-off service.

This synchronously releases the caller's previous request. If the last request is released the manager will initiate a transition to off, which can be observed by registering an *onoff_monitor*.

Note: Behavior is undefined if this is not paired with a preceding *onoff_request()* call that completed successfully.

Parameters

- `mgr` – the manager for which a request was successful.

Return values

- `non-negative` – the observed state (ONOFF_STATE_ON) of the machine at the time of the release, if the release succeeds.
- `-EIO` – if service has recorded an an error.
- `-ENOTSUP` – if the machine is not in a state that permits release.

int `onoff_cancel`(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Attempt to cancel an in-progress client operation.

It may be that a client has initiated an operation but needs to shut down before the operation has completed. For example, when a request was made and the need is no longer present.

Cancelling is supported only for *onoff_request()* and *onoff_reset()* operations, and is a synchronous operation. Be aware that any transition that was initiated on behalf of the client will continue to progress to completion: it is only notification of transition completion that may be eliminated. If there are no active requests when a transition to on completes the manager will initiate a transition to off.

Client notification does not occur for cancelled operations.

Parameters

- `mgr` – the manager for which an operation is to be cancelled.
- `cli` – a pointer to the same client state that was provided when the operation to be cancelled was issued.

Return values

- `non-negative` – the observed state of the machine at the time of the cancellation, if the cancellation succeeds. On successful cancellation ownership of `*cli` reverts to the client.
- `-EINVAL` – if the parameters are invalid.
- `-EALREADY` – if cli was not a record of an uncompleted notification at the time the cancellation was processed. This likely indicates that the operation and client notification had already completed.

static inline int `onoff_cancel_or_release`(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Helper function to safely cancel a request.

Some applications may want to issue requests on an asynchronous event (such as connection to a USB bus) and to release on a paired event (such as loss of connection to a USB bus). Applications cannot precisely determine that an in-progress request is still pending without using *onoff_monitor* and carefully avoiding race conditions.

This function is a helper that attempts to cancel the operation and issues a release if cancellation fails because the request was completed. This synchronously ensures that ownership of the client data reverts to the client so is available for a future request.

Parameters

- `mgr` – the manager for which an operation is to be cancelled.
- `cli` – a pointer to the same client state that was provided when *onoff_request()* was invoked. Behavior is undefined if this is a pointer to client data associated with an *onoff_reset()* request.

Return values

- `ONOFF_STATE_TO_ON` – if the cancellation occurred before the transition completed.

- ONOFF_STATE_ON – if the cancellation occurred after the transition completed.
- -EINVAL – if the parameters are invalid.
- negative – other errors produced by *onoff_release()*.

int onoff_reset(struct *onoff_manager* *mgr, struct *onoff_client* *cli)

Clear errors on an on-off service and reset it to its off state.

A service can only be reset when it is in an error state as indicated by *onoff_has_error()*.

The return value indicates the success or failure of an attempt to initiate an operation to reset the resource. If initiation of the operation succeeds the result of the reset operation itself is provided through the configured client notification method, possibly before this call returns. Multiple clients may request a reset; all are notified when it is complete.

Note that the call to this function may succeed in a case where the actual reset fails. Always check the operation completion result.

Note: Due to the conditions on state transition all incomplete asynchronous operations will have been informed of the error when it occurred. There need be no concern about dangling requests left after a reset completes.

Parameters

- mgr – the manager to be reset.
- cli – pointer to client state, including instructions on how to notify the client when reset completes. Behavior is undefined if cli references an object associated with an incomplete service operation.

Return values

- non-negative – the observed state of the machine at the time of the reset, if the reset succeeds.
- -ENOTSUP – if reset is not supported by the service.
- -EINVAL – if the parameters are invalid.
- -EALREADY – if the service does not have a recorded error.

int onoff_monitor_register(struct *onoff_manager* *mgr, struct *onoff_monitor* *mon)

Add a monitor of state changes for a manager.

Parameters

- mgr – the manager for which a state changes are to be monitored.
- mon – a linkable node providing a non-null callback to be invoked on state changes.

Returns non-negative on successful addition, or a negative error code.

int onoff_monitor_unregister(struct *onoff_manager* *mgr, struct *onoff_monitor* *mon)

Remove a monitor of state changes from a manager.

Parameters

- mgr – the manager for which a state changes are to be monitored.
- mon – a linkable node providing the callback to be invoked on state changes.

Returns non-negative on successful removal, or a negative error code.

int onoff_sync_lock(struct *onoff_sync_service* *srv, *k_spinlock_key_t* *keyp)

Lock a synchronous onoff service and provide its state.

Note: If an error state is returned it is the caller's responsibility to decide whether to preserve it (finalize with the same error state) or clear the error (finalize with a non-error result).

Parameters

- srv – pointer to the synchronous service state.
- keyp – pointer to where the lock key should be stored

Returns negative if the service is in an error state, otherwise the number of active requests at the time the lock was taken. The lock is held on return regardless of whether a negative state is returned.

int onoff_sync_finalize(struct *onoff_sync_service* *srv, *k_spinlock_key_t* key, struct *onoff_client* *cli, int res, bool on)

Process the completion of a transition in a synchronous service and release lock.

This function updates the service state on the res and on parameters then releases the lock. If cli is not null it finalizes the client notification using res.

If the service was in an error state when locked, and res is non-negative when finalized, the count is reset to zero before completing finalization.

Parameters

- srv – pointer to the synchronous service state
- key – the key returned by the preceding invocation of *onoff_sync_lock()*.
- cli – pointer to the onoff client through which completion information is returned. If a null pointer is passed only the state of the service is updated. For compatibility with the behavior of callbacks used with the manager API cli must be null when on is false (the manager does not support callbacks when turning off devices).
- res – the result of the transition. A negative value places the service into an error state. A non-negative value increments or decrements the reference count as specified by on.
- on – Only when res is non-negative, the service reference count will be incremented ifon is true, and decremented if on is false.

Returns negative if the service is left or put into an error state, otherwise the number of active requests at the time the lock was released.

struct onoff_transitions

#include <onoff.h> On-off service transition functions.

struct onoff_manager

#include <onoff.h> State associated with an on-off manager.

No fields in this structure are intended for use by service providers or clients. The state is to be initialized once, using *onoff_manager_init()*, when the service provider is initialized. In case of error it may be reset through the *onoff_reset()* API.

struct onoff_client

#include <onoff.h> State associated with a client of an on-off service.

Objects of this type are allocated by a client, which is responsible for zero-initializing the node field and invoking the approprite *sys_notify* init function to configure notification.

Control of the object content transfers to the service provider when a pointer to the object is passed to any on-off manager function. While the service provider controls the object the client must not change any object fields. Control reverts to the client concurrent with release of the owned *sys_notify* structure, or when indicated by an *onoff_cancel()* return value.

After control has reverted to the client the notify field must be reinitialized for the next operation.

Public Members

struct *sys_notify* `notify`

Notification configuration.

struct `onoff_monitor`

#include <onoff.h> Registration state for notifications of onoff service transitions.

Any given *onoff_monitor* structure can be associated with at most one *onoff_manager* instance.

Public Members

onoff_monitor_callback `callback`

Callback to be invoked on state change.

This must not be null.

struct `onoff_sync_service`

#include <onoff.h> State used when a driver uses the on-off service API for synchronous operations.

This is useful when a subsystem API uses the on-off API to support asynchronous operations but the transitions required by a particular driver are isr-ok and not sleep. It serves as a substitute for *onoff_manager*, with locking and persisted state updates supported by *onoff_sync_lock()* and *onoff_sync_finalize()*.

7.25 Shell

7.25.1 Overview

This module allows you to create and handle a shell with a user-defined command set. You can use it in examples where more than simple button or LED user interaction is required. This module is a Unix-like shell with these features:

- Support for multiple instances.
- Advanced cooperation with the *Logging*.
- Support for static and dynamic commands.
- Support for dictionary commands.
- Smart command completion with the `Tab` key.
- Built-in commands: `clear`, `shell`, `colors`, `echo`, `history` and `resize`.
- Viewing recently executed commands using keys: ↑ ↓ or meta keys.
- Text edition using keys: ←, →, `Backspace`, `Delete`, `End`, `Home`, `Insert`.
- Support for ANSI escape codes: `VT100` and `ESC[n~` for cursor control and color printing.
- Support for editing multiline commands.
- Built-in handler to display help for the commands.
- Support for wildcards: `*` and `?`.
- Support for meta keys.
- Support for getopt.
- Kconfig configuration to optimize memory usage.

Note: Some of these features have a significant impact on RAM and flash usage, but many can be disabled when not needed. To default to options which favor reduced RAM and flash requirements instead of features, you should enable `CONFIG_SHELL_MINIMAL` and selectively enable just the features you want.

The module can be connected to any transport for command input and output. At this point, the following transport layers are implemented:

- Segger RTT
- SMP
- Telnet
- UART
- USB

- DUMMY - not a physical transport layer.

Connecting to Segger RTT via TCP (on macOS, for example)

On macOS JLinkRTTClient won't let you enter input. Instead, please use following procedure:

- Open up a first Terminal window and enter:

```
JLinkRTTLogger -Device NRF52840_XXAA -RTTChannel 1 -if SWD -Speed 4000 ~/rtt.log
```

  (change device if required)

- Open up a second Terminal window and enter:

```
nc localhost 19021
```

- Now you should have a network connection to RTT that will let you enter input to the shell.

7.25.2 Commands

Shell commands are organized in a tree structure and grouped into the following types:

- Root command (level 0): Gathered and alphabetically sorted in a dedicated memory section.
- Static subcommand (level > 0): Number and syntax must be known during compile time. Created in the software module.
- Dynamic subcommand (level > 0): Number and syntax does not need to be known during compile time. Created in the software module.

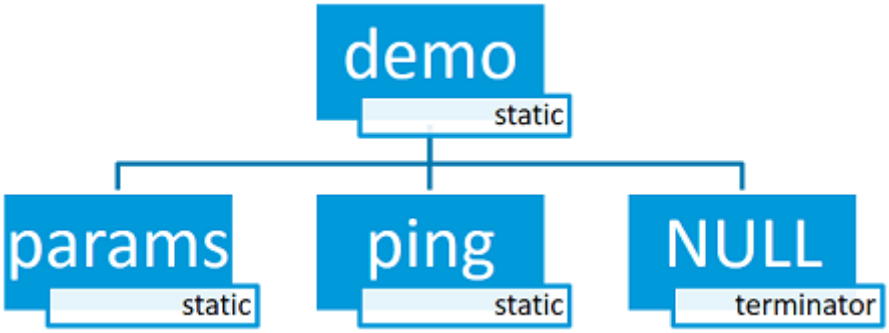
Creating commands

Use the following macros for adding shell commands:

- *SHELL_CMD_REGISTER* - Create root command. All root commands must have different name.
- *SHELL_COND_CMD_REGISTER* - Conditionally (if compile time flag is set) create root command. All root commands must have different name.
- *SHELL_CMD_ARG_REGISTER* - Create root command with arguments. All root commands must have different name.
- *SHELL_COND_CMD_ARG_REGISTER* - Conditionally (if compile time flag is set) create root command with arguments. All root commands must have different name.
- *SHELL_CMD* - Initialize a command.
- *SHELL_COND_CMD* - Initialize a command if compile time flag is set.
- *SHELL_EXPR_CMD* - Initialize a command if compile time expression is non-zero.
- *SHELL_CMD_ARG* - Initialize a command with arguments.
- *SHELL_COND_CMD_ARG* - Initialize a command with arguments if compile time flag is set.
- *SHELL_EXPR_CMD_ARG* - Initialize a command with arguments if compile time expression is non-zero.
- *SHELL_STATIC_SUBCMD_SET_CREATE* - Create a static subcommands array.
- *SHELL_SUBCMD_DICT_SET_CREATE* - Create a dictionary subcommands array.
- *SHELL_DYNAMIC_CMD_CREATE* - Create a dynamic subcommands array.

Commands can be created in any file in the system that includes include/shell/shell.h. All created commands are available for all shell instances.

Static commands Example code demonstrating how to create a root command with static subcommands.




```
/* Creating subcommands (level 1 command) array for command "demo". */
SHELL_STATIC_SUBCMD_SET_CREATE(sub_demo,
        SHELL_CMD(params, NULL, "Print params command.",
                                                cmd_demo_params),
        SHELL_CMD(ping,   NULL, "Ping command.", cmd_demo_ping),
        SHELL_SUBCMD_SET_END
);
/* Creating root (level 0) command "demo" */
SHELL_CMD_REGISTER(demo, &sub_demo, "Demo commands", NULL);
```

Example implementation can be found under following location: samples/subsys/shell/shell_module/src/main.c.

Dictionary commands

This is a special kind of static commands. Dictionary commands can be used every time you want to use a pair: (string <-> corresponding data) in a command handler. The string is usually a verbal description of a given data. The idea is to use the string as a command syntax that can be prompted by the shell and corresponding data can be used to process the command.

Let's use an example. Suppose you created a command to set an ADC gain. It is a perfect place where a dictionary can be used. The dictionary would be a set of pairs: (string: gain_value, int: value) where int value could be used with the ADC driver API.

Abstract code for this task would look like this:

```
static int gain_cmd_handler(const struct shell *shell,
                            size_t argc, char **argv, void *data)
{
        int gain;

        /* data is a value corresponding to called command syntax */
        gain = (int)data;
        adc_set_gain(gain);

        shell_print(shell, "ADC gain set to: %s\n"
                           "Value send to ADC driver: %d",
                           argv[0],
                           gain);

        return 0;
}
```

(continues on next page)

(continued from previous page)

```
SHELL_SUBCMD_DICT_SET_CREATE(sub_gain, gain_cmd_handler,
        (gain_1, 1), (gain_2, 2), (gain_1_2, 3), (gain_1_4, 4)
);
SHELL_CMD_REGISTER(gain, &sub_gain, "Set ADC gain", NULL);
```

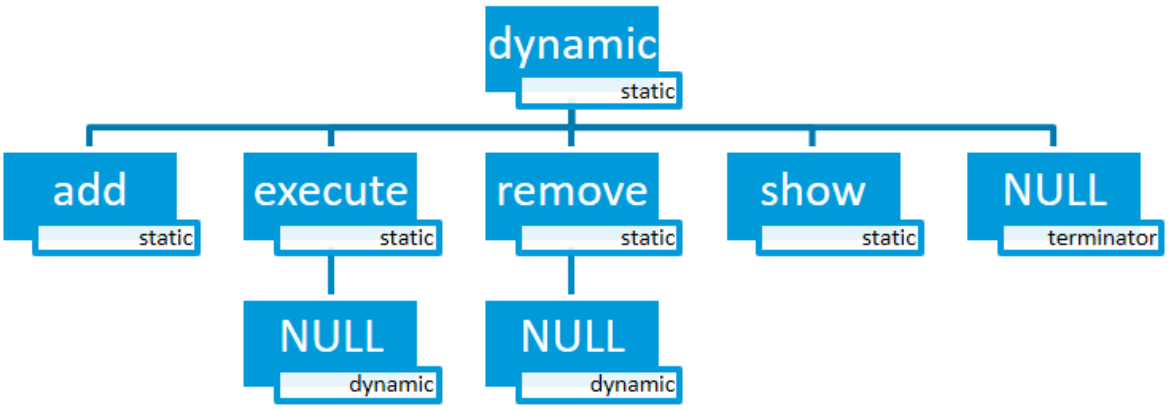
This is how it would look like in the shell:

```
uart:~$ gain ga
  gain_1    gain_2    gain_1_2  gain_1_4
uart:~$ gain gain_1
ADC gain set to: gain_1
Value send to ADC driver: 1
uart:~$ gain gain_2
ADC gain set to: gain_2
Value send to ADC driver: 2
uart:~$ gain gain_1_2
ADC gain set to: gain_1_2
Value send to ADC driver: 3
uart:~$ gain gain_1_4
ADC gain set to: gain_1_4
Value send to ADC driver: 4
uart:~$
```

Dynamic commands Example code demonstrating how to create a root command with static and dynamic subcommands. At the beginning dynamic command list is empty. New commands can be added by typing:

```
dynamic add <new_dynamic_command>
```

Newly added commands can be prompted or autocompleted with the `Tab` key.




```
/* Buffer for 10 dynamic commands */
static char dynamic_cmd_buffer[10][50];

/* commands counter */
static uint8_t dynamic_cmd_cnt;

/* Function returning command dynamically created
```

(continues on next page)

(continued from previous page)

```
 * in  dynamic_cmd_buffer.
 */
static void dynamic_cmd_get(size_t idx,
                            struct shell_static_entry *entry)
{
        if (idx < dynamic_cmd_cnt) {
                entry->syntax = dynamic_cmd_buffer[idx];
                entry->handler  = NULL;
                entry->subcmd = NULL;
                entry->help = "Show dynamic command name.";
        } else {
                /* if there are no more dynamic commands available
                 * syntax must be set to NULL.
                 */
                entry->syntax = NULL;
        }
}

SHELL_DYNAMIC_CMD_CREATE(m_sub_dynamic_set, dynamic_cmd_get);
SHELL_STATIC_SUBCMD_SET_CREATE(m_sub_dynamic,
        SHELL_CMD(add, NULL,"Add new command to dynamic_cmd_buffer and"
                  " sort them alphabetically.",
                  cmd_dynamic_add),
        SHELL_CMD(execute, &m_sub_dynamic_set,
                  "Execute a command.", cmd_dynamic_execute),
        SHELL_CMD(remove, &m_sub_dynamic_set,
                  "Remove a command from dynamic_cmd_buffer.",
                  cmd_dynamic_remove),
        SHELL_CMD(show, NULL,
                  "Show all commands in dynamic_cmd_buffer.",
                  cmd_dynamic_show),
        SHELL_SUBCMD_SET_END
);
SHELL_CMD_REGISTER(dynamic, &m_sub_dynamic,
           "Demonstrate dynamic command usage.", cmd_dynamic);
```

Example implementation can be found under following location: samples/subsys/shell/shell_module/src/dynamic_cmd.c.

Commands execution

Each command or subcommand may have a handler. The shell executes the handler that is found deepest in the command tree and further subcommands (without a handler) are passed as arguments. Characters within parentheses are treated as one argument. If shell wont find a handler it will display an error message.

Commands can be also executed from a user application using any active backend and a function *shell_execute_cmd()*, as shown in this example:

```
void main(void)
{
        /* Below code will execute "clear" command on a DUMMY backend */
        shell_execute_cmd(NULL, "clear");

        /* Below code will execute "shell colors off" command on
         * an UART backend
```

(continues on next page)

(continued from previous page)

```
         */
        shell_execute_cmd(shell_backend_uart_get_ptr(),
                          "shell colors off");
}
```

Enable the DUMMY backend by setting the Kconfig `CONFIG_SHELL_BACKEND_DUMMY` option.

Command handler Simple command handler implementation:

```
static int cmd_handler(const struct shell *shell, size_t argc,
                       char **argv)
{
        ARG_UNUSED(argc);
        ARG_UNUSED(argv);

        shell_fprintf(shell, SHELL_INFO, "Print info message\n");

        shell_print(shell, "Print simple text.");

        shell_warn(shell, "Print warning text.");

        shell_error(shell, "Print error text.");

        return 0;
}
```

Function *shell_fprintf()* or the shell print macros: *shell_print*, *shell_info*, *shell_warn* and *shell_error* can be used from the command handler or from threads, but not from an interrupt context. Instead, interrupt handlers should use *Logging* for printing.

Command help Every user-defined command or subcommand can have its own help description. The help for commands and subcommands can be created with respective macros: *SHELL_CMD_REGISTER*, *SHELL_CMD_ARG_REGISTER*, *SHELL_CMD*, and *SHELL_CMD_ARG*.

Shell prints this help message when you call a command or subcommand with `-h` or `--help` parameter.

Parent commands In the subcommand handler, you can access both the parameters passed to commands or the parent commands, depending on how you index `argv`.

- When indexing `argv` with positive numbers, you can access the parameters.
- When indexing `argv` with negative numbers, you can access the parent commands.
- The subcommand to which the handler belongs has the `argv` index of 0.

```
static int cmd_handler(const struct shell *shell, size_t argc,
                       char **argv)
{
        ARG_UNUSED(argc);

        /* If it is a subcommand handler parent command syntax
         * can be found using argv[-1].
         */
        shell_print(shell, "This command has a parent command: %s",
                    argv[-1]);
```

(continues on next page)

(continued from previous page)

```
        /* Print this command syntax */
        shell_print(shell, "This command syntax is: %s", argv[0]);

        /* Print first argument */
        shell_print(shell, "%s", argv[1]);

        return 0;
}
```

Built-in commands

These commands are activated by `CONFIG_SHELL_CMDS` set to `y`.

- `clear` - Clears the screen.
- `history` - Shows the recently entered commands.
- `resize` - Must be executed when terminal width is different than 80 characters or after each change of terminal width. It ensures proper multiline text display and ←, →, `End`, `Home` keys handling. Currently this command works only with UART flow control switched on. It can be also called with a subcommand:
  - `default` - Shell will send terminal width = 80 to the terminal and assume successful delivery.

  These command needs extra activation: `CONFIG_SHELL_CMDS_RESIZE` set to `y`.
- `select` - It can be used to set new root command. Exit to main command tree is with alt+r. This command needs extra activation: `CONFIG_SHELL_CMDS_SELECT` set to `y`.
- `shell` - Root command with useful shell-related subcommands like:
  - `echo` - Toggles shell echo.
  - `colors` - Toggles colored syntax. This might be helpful in case of Bluetooth shell to limit the amount of transferred bytes.
  - `stats` - Shows shell statistics.

7.25.3 Tab Feature

The Tab button can be used to suggest commands or subcommands. This feature is enabled by `CONFIG_SHELL_TAB` set to `y`. It can also be used for partial or complete auto-completion of commands. This feature is activated by `CONFIG_SHELL_TAB_AUTOCOMPLETION` set to `y`. When user starts writing a command and presses the `Tab` button then the shell will do one of 3 possible things:

- Autocomplete the command.
- Prompts available commands and if possible partly completes the command.
- Will not do anything if there are no available or matching commands.

```
uart:~$ log
  backend        disable        enable         go             halt
  list_backends  status
uart:~$ log enable
  dbg   err   inf   none  wrn
uart:~$ log enable err
  app               app_test          log              os
  shell.shell_uart  shell_uart
uart:~$ log enable err shell
  shell.shell_uart  shell_uart
uart:~$ log enable err shell
```

7.25.4 History Feature

This feature enables commands history in the shell. It is activated by: `CONFIG_SHELL_HISTORY` set to `y`. History can be accessed using keys: ↑ ↓ or `Ctrl + n` and `Ctrl + p` if meta keys are active. Number of commands that can be stored depends on size of `CONFIG_SHELL_HISTORY_BUFFER` parameter.

7.25.5 Wildcards Feature

The shell module can handle wildcards. Wildcards are interpreted correctly when expanded command and its subcommands do not have a handler. For example, if you want to set logging level to `err` for the `app` and `app_test` modules you can execute the following command:

```
log enable err a*
```

```
uart:~$ log status
module_name                                     | current | built-in
------------------------------------------------------------------
app                                             | inf     | inf
app_test                                        | inf     | inf
shell.uart_shell                                | inf     | inf
uart:~$ log enable err a*
uart:~$ log status
module_name                                     | current | built-in
------------------------------------------------------------------
app                                             | err     | inf
app_test                                        | err     | inf
shell.uart_shell                                | inf     | inf
uart:~$
```

This feature is activated by `CONFIG_SHELL_WILDCARD` set to `y`.

7.25.6 Meta Keys Feature

The shell module supports the following meta keys:

Table 7: Implemented meta keys

Meta keys	Action
`Ctrl + a`	Moves the cursor to the beginning of the line.
`Ctrl + b`	Moves the cursor backward one character.
`Ctrl + c`	Preserves the last command on the screen and starts a new command in a new line.
`Ctrl + d`	Deletes the character under the cursor.
`Ctrl + e`	Moves the cursor to the end of the line.
`Ctrl + f`	Moves the cursor forward one character.
`Ctrl + k`	Deletes from the cursor to the end of the line.
`Ctrl + l`	Clears the screen and leaves the currently typed command at the top of the screen.
`Ctrl + n`	Moves in history to next entry.
`Ctrl + p`	Moves in history to previous entry.
`Ctrl + u`	Clears the currently typed command.
`Ctrl + w`	Removes the word or part of the word to the left of the cursor. Words separated by period instead of space are treated as one word.
`Alt + b`	Moves the cursor backward one word.
`Alt + f`	Moves the cursor forward one word.

This feature is activated by `CONFIG_SHELL_METAKEYS` set to `y`.

7.25.7 Getopt Feature

Some shell users apart from subcommands might need to use options as well. the arguments string, looking for supported options. Typically, this task is accomplished by the `getopt` function.

For this purpose shell supports the getopt library available in the FreeBSD project. I was modified so that it can be used by all instances of the shell at the same time, hence its call requires one more parameter.

An example usage:

```
while ((char c = shell_getopt(shell, argc, argv, "abhc:")) != -1) {
   /* some code */
}
```

This module is activated by `CONFIG_SHELL_GETOPT` set to `y`.

7.25.8 Obscured Input Feature

With the obscured input feature, the shell can be used for implementing a login prompt or other user interaction whereby the characters the user types should not be revealed on screen, such as when entering a password.

Once the obscured input has been accepted, it is normally desired to return the shell to normal operation. Such runtime control is possible with the `shell_obscure_set` function.

An example of login and logout commands using this feature is located in samples/subsys/shell/shell_module/src/main.c and the config file samples/subsys/shell/shell_module/prj_login.conf.

This feature is activated upon startup by `CONFIG_SHELL_START_OBSCURED` set to `y`. With this set either way, the option can still be controlled later at runtime. `CONFIG_SHELL_CMDS_SELECT` is useful to prevent entry of any other command besides a login command, by means of the `shell_set_root_cmd` function. Likewise, `CONFIG_SHELL_PROMPT_UART` allows you to set the prompt upon startup, but it can be changed later with the `shell_prompt_change` function.

7.25.9 Shell Logger Backend Feature

Shell instance can act as the *Logging* backend. Shell ensures that log messages are correctly multiplexed with shell output. Log messages from logger thread are enqueued and processed in the shell thread. Logger thread will block for configurable amount of time if queue is full, blocking logger thread context for that time. Oldest log message is removed from the queue after timeout and new message is enqueued. Use the `shell stats show` command to retrieve number of log messages dropped by the shell instance. Log queue size and timeout are *SHELL_DEFINE* arguments.

This feature is activated by: `CONFIG_SHELL_LOG_BACKEND` set to `y`.

Warning: Enqueuing timeout must be set carefully when multiple backends are used in the system. The shell instance could have a slow transport or could block, for example, by a UART with hardware flow control. If timeout is set too high, the logger thread could be blocked and impact other logger backends.

Warning: As the shell is a complex logger backend, it can not output logs if the application crashes before the shell thread is running. In this situation, you can enable one of the simple logging backends instead, such as UART (`CONFIG_LOG_BACKEND_UART`) or RTT (`CONFIG_LOG_BACKEND_RTT`), which are available earlier during system initialization.

7.25.10 Usage

To create a new shell instance user needs to activate requested backend using `menuconfig`.

The following code shows a simple use case of this library:

```
void main(void)
{

}

static int cmd_demo_ping(const struct shell *shell, size_t argc,
                         char **argv)
{
        ARG_UNUSED(argc);
        ARG_UNUSED(argv);

        shell_print(shell, "pong");
        return 0;
}

static int cmd_demo_params(const struct shell *shell, size_t argc,
                           char **argv)
{
        int cnt;

        shell_print(shell, "argc = %d", argc);
        for (cnt = 0; cnt < argc; cnt++) {
                shell_print(shell, "  argv[%d] = %s", cnt, argv[cnt]);
        }
        return 0;
}
```

(continues on next page)

(continued from previous page)

```
/* Creating subcommands (level 1 command) array for command "demo". */
SHELL_STATIC_SUBCMD_SET_CREATE(sub_demo,
        SHELL_CMD(params, NULL, "Print params command.",
                                                cmd_demo_params),
        SHELL_CMD(ping,   NULL, "Ping command.", cmd_demo_ping),
        SHELL_SUBCMD_SET_END
);
/* Creating root (level 0) command "demo" without a handler */
SHELL_CMD_REGISTER(demo, &sub_demo, "Demo commands", NULL);

/* Creating root (level 0) command "version" */
SHELL_CMD_REGISTER(version, NULL, "Show kernel version", cmd_version);
```

Users may use the `Tab` key to complete a command/subcommand or to see the available subcommands for the currently entered command level. For example, when the cursor is positioned at the beginning of the command line and the `Tab` key is pressed, the user will see all root (level 0) commands:

```
clear  demo  shell  history  log  resize  version
```

Note: To view the subcommands that are available for a specific command, you must first type a `space` after this command and then hit `Tab`.

These commands are registered by various modules, for example:

- `clear`, `shell`, `history`, and `resize` are built-in commands which have been registered by subsys/shell/shell.c
- `demo` and `version` have been registered in example code above by main.c
- `log` has been registered by subsys/logging/log_cmds.c

Then, if a user types a `demo` command and presses the `Tab` key, the shell will only print the subcommands registered for this command:

```
params  ping
```

7.25.11 API Reference

group `shell_api`

Shell API.

Defines

`SHELL_CMD_ARG_REGISTER(syntax, subcmd, help, handler, mandatory, optional)`

Macro for defining and adding a root command (level 0) with required number of arguments.

Note: Each root command shall have unique syntax. If a command will be called with wrong number of arguments shell will print an error message and command handler will not be called.

Parameters

- `syntax` – **[in]** Command syntax (for example: history).

- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.
- `mandatory` – **[in]** Number of mandatory arguments includig command name.
- `optional` – **[in]** Number of optional arguments.

`SHELL_COND_CMD_ARG_REGISTER(flag, syntax, subcmd, help, handler, mandatory, optional)`

Macro for defining and adding a conditional root command (level 0) with required number of arguments.

Macro can be used to create a command which can be conditionally present. It is and alternative to #ifdefs around command registration and command handler. If command is disabled handler and subcommands are removed from the application.

See also:

SHELL_CMD_ARG_REGISTER for details.

Parameters

- `flag` – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.
- `mandatory` – **[in]** Number of mandatory arguments includig command name.
- `optional` – **[in]** Number of optional arguments.

`SHELL_CMD_REGISTER(syntax, subcmd, help, handler)`

Macro for defining and adding a root command (level 0) with arguments.

Note: All root commands must have different name.

Parameters

- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.

`SHELL_COND_CMD_REGISTER(flag, syntax, subcmd, help, handler)`

Macro for defining and adding a conditional root command (level 0) with arguments.

See also:

SHELL_COND_CMD_ARG_REGISTER.

Parameters

- `flag` – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.

`SHELL_STATIC_SUBCMD_SET_CREATE(name, ...)`

Macro for creating a subcommand set. It must be used outside of any function body.

Example usage: SHELL_STATIC_SUBCMD_SET_CREATE(foo, *SHELL_CMD(abc, ...)*, *SHELL_CMD(def, ...)*, SHELL_SUBCMD_SET_END)

Parameters

- `name` – **[in]** Name of the subcommand set.
- `...` – **[in]** List of commands created with *SHELL_CMD_ARG* or or *SHELL_CMD*

`SHELL_SUBCMD_SET_END`

Define ending subcommands set.

`SHELL_DYNAMIC_CMD_CREATE(name, get)`

Macro for creating a dynamic entry.

Parameters

- `name` – **[in]** Name of the dynamic entry.
- `get` – **[in]** Pointer to the function returning dynamic commands array

`SHELL_CMD_ARG(syntax, subcmd, help, handler, mand, opt)`

Initializes a shell command with arguments.

Note: If a command will be called with wrong number of arguments shell will print an error message and command handler will not be called.

Parameters

- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.
- `mand` – **[in]** Number of mandatory arguments includig command name.
- `opt` – **[in]** Number of optional arguments.

`SHELL_COND_CMD_ARG(flag, syntax, subcmd, help, handler, mand, opt)`

Initializes a conditional shell command with arguments.

See also:

SHELL_CMD_ARG. Based on the flag, creates a valid entry or an empty command which is ignored by the *shell*. It is an alternative to #ifdefs around command registration and command handler. However, empty structure is present in the flash even if command is disabled (subcommands and handler are removed). Macro internally handles case if flag is not defined

so flag must be provided without any wrapper, e.g.: *SHELL_COND_CMD_ARG(CONFIG_FOO, ...)*

Parameters

- `flag` – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- `syntax` – **[in]** Command syntax (for example: history).
- `subcmd` – **[in]** Pointer to a subcommands array.
- `help` – **[in]** Pointer to a command help string.
- `handler` – **[in]** Pointer to a function handler.
- `mand` – **[in]** Number of mandatory arguments includig command name.
- `opt` – **[in]** Number of optional arguments.

`SHELL_EXPR_CMD_ARG(_expr, _syntax, _subcmd, _help, _handler, _mand, _opt)`

Initializes a conditional shell command with arguments if expression gives non-zero result at compile time.

See also:

SHELL_CMD_ARG. Based on the expression, creates a valid entry or an empty command which is ignored by the *shell*. It should be used instead of *SHELL_COND_CMD_ARG* if condition is not a single configuration flag, e.g.: *SHELL_EXPR_CMD_ARG*(*IS_ENABLED(CONFIG_FOO)* && *IS_ENABLED(CONFIG_FOO_SETTING_1)*, ...)

Parameters

- `_expr` – **[in]** Expression.
- `_syntax` – **[in]** Command syntax (for example: history).
- `_subcmd` – **[in]** Pointer to a subcommands array.
- `_help` – **[in]** Pointer to a command help string.
- `_handler` – **[in]** Pointer to a function handler.
- `_mand` – **[in]** Number of mandatory arguments includig command name.
- `_opt` – **[in]** Number of optional arguments.

`SHELL_CMD(_syntax, _subcmd, _help, _handler)`

Initializes a shell command.

Parameters

- `_syntax` – **[in]** Command syntax (for example: history).
- `_subcmd` – **[in]** Pointer to a subcommands array.
- `_help` – **[in]** Pointer to a command help string.
- `_handler` – **[in]** Pointer to a function handler.

`SHELL_COND_CMD(_flag, _syntax, _subcmd, _help, _handler)`

Initializes a conditional shell command.

See also:

SHELL_COND_CMD_ARG.

Parameters

- `_flag` – **[in]** Compile time flag. Command is present only if flag exists and equals 1.
- `_syntax` – **[in]** Command syntax (for example: history).
- `_subcmd` – **[in]** Pointer to a subcommands array.
- `_help` – **[in]** Pointer to a command help string.
- `_handler` – **[in]** Pointer to a function handler.

`SHELL_EXPR_CMD(_expr, _syntax, _subcmd, _help, _handler)`

Initializes shell command if expression gives non-zero result at compile time.

See also:

SHELL_EXPR_CMD_ARG.

Parameters

- `_expr` – **[in]** Compile time expression. Command is present only if expression is non-zero.
- `_syntax` – **[in]** Command syntax (for example: history).
- `_subcmd` – **[in]** Pointer to a subcommands array.
- `_help` – **[in]** Pointer to a command help string.
- `_handler` – **[in]** Pointer to a function handler.

`SHELL_CMD_DICT_CREATE(_data)`

`SHELL_SUBCMD_DICT_SET_CREATE(_name, _handler, ...)`

Initializes shell dictionary commands.

This is a special kind of static commands. Dictionary commands can be used every time you want to use a pair: (string <-> corresponding data) in a command handler. The string is usually a verbal description of a given data. The idea is to use the string as a command syntax that can be prompted by the shell and corresponding data can be used to process the command.

Example usage: static int my_handler(const struct shell *shell,

size_t argc, char **argv, void *data) { int val = (int)data;

See also:

shell_dict_cmd_handler

shell_print(shell, "(syntax, value) : (%s, %d)", argv[0], val); return 0; }

SHELL_SUBCMD_DICT_SET_CREATE(sub_dict_cmds, my_handler, (value_0, 0), (value_1, 1), (value_2, 2), (value_3, 3)); *SHELL_CMD_REGISTER(dictionary, &sub_dict_cmds, NULL, NULL)*;

Parameters

- `_name` – **[in]** Name of the dictionary subcommand set
- `_handler` – **[in]** Command handler common for all dictionary commands.
- `...` – **[in]** Dictionary pairs: (command_syntax, value). Value will be passed to the _handler as user data.

SHELL_DEFINE(_name, _prompt, _transport_iface, _log_queue_size, _log_timeout, _shell_flag)

Macro for defining a shell instance.

Parameters

- _name – **[in]** Instance name.
- _prompt – **[in]** Shell default prompt string.
- _transport_iface – **[in]** Pointer to the transport interface.
- _log_queue_size – **[in]** Logger processing queue size.
- _log_timeout – **[in]** Logger thread timeout in milliseconds on full log queue. If queue is full logger thread is blocked for given amount of time before log message is dropped.
- _shell_flag – **[in]** Shell output newline sequence.

SHELL_NORMAL

Terminal default text color for shell_fprintf function.

SHELL_INFO

Green text color for shell_fprintf function.

SHELL_OPTION

Cyan text color for shell_fprintf function.

SHELL_WARNING

Yellow text color for shell_fprintf function.

SHELL_ERROR

Red text color for shell_fprintf function.

shell_info(_sh, _ft, ...)

Print info message to the shell.

See shell_fprintf.

Parameters

- _sh – **[in]** Pointer to the shell instance.
- _ft – **[in]** Format string.
- ... – **[in]** List of parameters to print.

shell_print(_sh, _ft, ...)

Print normal message to the shell.

See shell_fprintf.

Parameters

- _sh – **[in]** Pointer to the shell instance.
- _ft – **[in]** Format string.
- ... – **[in]** List of parameters to print.

shell_warn(_sh, _ft, ...)

Print warning message to the shell.

See shell_fprintf.

Parameters

- `_sh` – **[in]** Pointer to the shell instance.
- `_ft` – **[in]** Format string.
- `...` – **[in]** List of parameters to print.

`shell_error(_sh, _ft, ...)`

Print error message to the shell.

See shell_fprintf.

Parameters

- `_sh` – **[in]** Pointer to the shell instance.
- `_ft` – **[in]** Format string.
- `...` – **[in]** List of parameters to print.

`SHELL_CMD_HELP_PRINTED`

Typedefs

typedef void (*`shell_dynamic_get`)(size_t idx, struct *shell_static_entry* *entry)

Shell dynamic command descriptor.

Function shall fill the received *shell_static_entry* structure with requested (idx) dynamic subcommand data. If there is more than one dynamic subcommand available, the function shall ensure that the returned commands: entry->syntax are sorted in alphabetical order. If idx exceeds the available dynamic subcommands, the function must write to entry->syntax NULL value. This will indicate to the shell module that there are no more dynamic commands to read.

typedef int (*`shell_cmd_handler`)(const struct *shell* *shell, size_t argc, char **argv)

Shell command handler prototype.

Param shell Shell instance.

Param argc Arguments count.

Param argv Arguments.

Retval 0 Successful command execution.

Retval 1 Help printed and command not executed.

Retval -EINVAL Argument validation failed.

Retval -ENOEXEC Command not executed.

typedef int (*`shell_dict_cmd_handler`)(const struct *shell* *shell, size_t argc, char **argv, void *data)

Shell dictionary command handler prototype.

Param shell Shell instance.

Param argc Arguments count.

Param argv Arguments.

Param data Pointer to the user data.

Retval 0 Successful command execution.

Retval 1 Help printed and command not executed.

Retval -EINVAL Argument validation failed.

Retval -ENOEXEC Command not executed.

typedef void (*`shell_transport_handler_t`)(enum *shell_transport_evt* evt, void *context)

typedef void (*`shell_uninit_cb_t`)(const struct *shell* *shell, int res)

typedef void (*`shell_bypass_cb_t`)(const struct *shell* *shell, uint8_t *data, size_t len)

Bypass callback.

Param shell Shell instance.

Param data Raw data from transport.

Param len Data length.

Enums

enum `shell_receive_state`

Values:

enumerator `SHELL_RECEIVE_DEFAULT`

enumerator `SHELL_RECEIVE_ESC`

enumerator `SHELL_RECEIVE_ESC_SEQ`

enumerator `SHELL_RECEIVE_TILDE_EXP`

enum `shell_state`

Values:

enumerator `SHELL_STATE_UNINITIALIZED`

enumerator `SHELL_STATE_INITIALIZED`

enumerator `SHELL_STATE_ACTIVE`

enumerator `SHELL_STATE_PANIC_MODE_ACTIVE`

Panic activated.

enumerator `SHELL_STATE_PANIC_MODE_INACTIVE`

Panic requested, not supported.

enum `shell_transport_evt`

Shell transport event.

Values:

enumerator SHELL_TRANSPORT_EVT_RX_RDY

enumerator SHELL_TRANSPORT_EVT_TX_RDY

enum shell_signal

Values:

enumerator SHELL_SIGNAL_RXRDY

enumerator SHELL_SIGNAL_LOG_MSG

enumerator SHELL_SIGNAL_KILL

enumerator SHELL_SIGNAL_TXDONE

enumerator SHELL_SIGNALS

enum shell_flag

Flags for setting shell output newline sequence.

Values:

enumerator SHELL_FLAG_CRLF_DEFAULT = (1 << 0)

enumerator SHELL_FLAG_OLF_CRLF = (1 << 1)

Functions

const struct *device* *shell_device_lookup(size_t idx, const char *prefix)

Get by index a device that matches .

This can be used, for example, to identify I2C_1 as the second I2C device.

Devices that failed to initialize or do not have a non-empty name are excluded from the candidates for a match.

Parameters

- idx – the device number starting from zero.
- prefix – optional name prefix used to restrict candidate devices. Indexing is done relative to devices with names that start with this text. Pass null if no prefix match is required.

int shell_init(const struct *shell* *shell, const void *transport_config, bool use_colors, bool log_backend, uint32_t init_log_level)

Function for initializing a transport layer and internal shell state.

Parameters

- shell – **[in]** Pointer to shell instance.
- transport_config – **[in]** Transport configuration during initialization.
- use_colors – **[in]** Enables colored prints.

- `log_backend` – If true, the console will be used as logger backend.
- `init_log_level` – **[in]** Default severity level for the logger.

Returns Standard error code.

void `shell_uninit`(const struct *shell* *shell, *shell_uninit_cb_t* cb)

Uninitializes the transport layer and the internal shell state.

Parameters

- `shell` – Pointer to shell instance.
- `cb` – Callback called when uninitialization is completed.

Returns Standard error code.

int `shell_start`(const struct *shell* *shell)

Function for starting shell processing.

Parameters

- `shell` – Pointer to the shell instance.

Returns Standard error code.

int `shell_stop`(const struct *shell* *shell)

Function for stopping shell processing.

Parameters

- `shell` – Pointer to shell instance.

Returns Standard error code.

void `shell_fprintf`(const struct *shell* *shell, enum shell_vt100_color color, const char *fmt, ...)

printf-like function which sends formatted data stream to the shell.

This function can be used from the command handler or from threads, but not from an interrupt context.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `color` – **[in]** Printed text color.
- `fmt` – **[in]** Format string.
- `...` – **[in]** List of parameters to print.

void `shell_vfprintf`(const struct *shell* *shell, enum shell_vt100_color color, const char *fmt, va_list args)

vprintf-like function which sends formatted data stream to the shell.

This function can be used from the command handler or from threads, but not from an interrupt context. It is similar to *shell_fprintf()* but takes a va_list instead of variable arguments.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `color` – **[in]** Printed text color.
- `fmt` – **[in]** Format string.
- `args` – **[in]** List of parameters to print.

void `shell_hexdump_line`(const struct *shell* *shell, unsigned int offset, const uint8_t *data, size_t len)

Print a line of data in hexadecimal format.

Each line shows the offset, bytes and then ASCII representation.

For example:

00008010: 20 25 00 20 2f 48 00 08 80 05 00 20 af 46 00 | %. /H.. ... .F. |

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `offset` – **[in]** Offset to show for this line.
- `data` – **[in]** Pointer to data.
- `len` – **[in]** Length of data.

void `shell_hexdump`(const struct *shell* *shell, const uint8_t *data, size_t len)

Print data in hexadecimal format.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `data` – **[in]** Pointer to data.
- `len` – **[in]** Length of data.

void `shell_process`(const struct *shell* *shell)

Process function, which should be executed when data is ready in the transport interface. To be used if shell thread is disabled.

Parameters

- `shell` – **[in]** Pointer to the shell instance.

int `shell_prompt_change`(const struct *shell* *shell, const char *prompt)

Change displayed shell prompt.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `prompt` – **[in]** New shell prompt.

Returns 0 Success.

Returns -EINVAL Pointer to new prompt is not correct.

void `shell_help`(const struct *shell* *shell)

Prints the current command help.

Function will print a help string with: the currently entered command and subcommands (if they exist).

Parameters

- `shell` – **[in]** Pointer to the shell instance.

int `shell_execute_cmd`(const struct *shell* *shell, const char *cmd)

Execute command.

Pass command line to shell to execute.

Note: This by no means makes any of the commands a stable interface, so this function should only be used for debugging/diagnostic.

This function must not be called from shell command context!

Parameters

- `shell` – **[in]** Pointer to the shell instance. It can be NULL when the `CONFIG_SHELL_BACKEND_DUMMY` option is enabled.
- `cmd` – **[in]** Command to be executed.

Returns Result of the execution

int `shell_set_root_cmd`(const char *cmd)

Set root command for all shell instances.

It allows setting from the code the root command. It is an equivalent of calling select command with one of the root commands as the argument (e.g "select log") except it sets command for all shell instances.

Parameters

- `cmd` – String with one of the root commands or null pointer to reset.

Return values

- `0` – if root command is set.
- `-EINVAL` – if invalid root command is provided.

void `shell_set_bypass`(const struct *shell* *shell, *shell_bypass_cb_t* bypass)

Set bypass callback.

Bypass callback is called whenever data is received. Shell is bypassed and data is passed directly to the callback. Use null to disable bypass functionality.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `bypass` – **[in]** Bypass callback or null to disable.

int `shell_insert_mode_set`(const struct *shell* *shell, bool val)

Allow application to control text insert mode. Value is modified atomically and the previous value is returned.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `val` – **[in]** Insert mode.

Return values

- `0` – or 1: previous value
- `-EINVAL` – if shell is NULL.

int `shell_use_colors_set`(const struct *shell* *shell, bool val)

Allow application to control whether terminal output uses colored syntax. Value is modified atomically and the previous value is returned.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `val` – **[in]** Color mode.

Return values

- `0` – or 1: previous value
- `-EINVAL` – if shell is NULL.

int `shell_echo_set`(const struct *shell* *shell, bool val)

Allow application to control whether user input is echoed back. Value is modified atomically and the previous value is returned.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `val` – **[in]** Echo mode.

Return values

- `0` – or 1: previous value
- `-EINVAL` – if shell is NULL.

int `shell_obscure_set`(const struct *shell* *shell, bool obscure)

Allow application to control whether user input is obscured with asterisks — useful for implementing passwords. Value is modified atomically and the previous value is returned.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `obscure` – **[in]** Obscure mode.

Return values

- `0` – or 1: previous value.
- `-EINVAL` – if shell is NULL.

int `shell_mode_delete_set`(const struct *shell* *shell, bool val)

Allow application to control whether the delete key backspaces or deletes. Value is modified atomically and the previous value is returned.

Parameters

- `shell` – **[in]** Pointer to the shell instance.
- `val` – **[in]** Delete mode.

Return values

- `0` – or 1: previous value
- `-EINVAL` – if shell is NULL.

Variables

const struct *log_backend_api* `log_backend_shell_api`

struct `shell_cmd_entry`

#include <shell.h> Shell command descriptor.

union `union_cmd_entry`

#include <shell.h>

Public Members

shell_dynamic_get dynamic_get

< Pointer to function returning dynamic commands. Pointer to array of static commands.

const struct *shell_static_entry* *entry

struct shell_static_args

#include <shell.h>

Public Members

uint8_t mandatory

Number of mandatory arguments.

uint8_t optional

Number of optional arguments.

struct shell_static_entry

#include <shell.h>

Public Members

const char *syntax

Command syntax strings.

const char *help

Command help string.

const struct *shell_cmd_entry* *subcmd

Pointer to subcommand.

shell_cmd_handler handler

Command handler.

struct *shell_static_args* args

Command arguments.

struct shell_transport_api

#include <shell.h> Unified shell transport interface.

Public Members

int (*init)(const struct *shell_transport* *transport, const void *config, *shell_transport_handler_t* evt_handler, void *context)

Function for initializing the shell transport interface.

Param transport [in] Pointer to the transfer instance.

Param config [in] Pointer to instance configuration.

Param evt_handler **[in]** Event handler.
Param context **[in]** Pointer to the context passed to event handler.
Return Standard error code.

int (*`uninit`)(const struct *shell_transport* *transport)

Function for uninitializing the shell transport interface.
Param transport **[in]** Pointer to the transfer instance.
Return Standard error code.

int (*`enable`)(const struct *shell_transport* *transport, bool blocking_tx)

Function for enabling transport in given TX mode.

Function can be used to reconfigure TX to work in blocking mode.
Param transport Pointer to the transfer instance.
Param blocking_tx If true, the transport TX is enabled in blocking mode.
Return NRF_SUCCESS on successful enabling, error otherwise (also if not supported).

int (*`write`)(const struct *shell_transport* *transport, const void *data, size_t length, size_t *cnt)

Function for writing data to the transport interface.
Param transport **[in]** Pointer to the transfer instance.
Param data **[in]** Pointer to the source buffer.
Param length **[in]** Source buffer length.
Param cnt **[out]** Pointer to the sent bytes counter.
Return Standard error code.

int (*`read`)(const struct *shell_transport* *transport, void *data, size_t length, size_t *cnt)

Function for reading data from the transport interface.
Param p_transport **[in]** Pointer to the transfer instance.
Param p_data **[in]** Pointer to the destination buffer.
Param length **[in]** Destination buffer length.
Param cnt **[out]** Pointer to the received bytes counter.
Return Standard error code.

void (*`update`)(const struct *shell_transport* *transport)

Function called in shell thread loop.

Can be used for backend operations that require longer execution time
Param transport **[in]** Pointer to the transfer instance.

struct `shell_transport`

#include <shell.h>

struct `shell_stats`

#include <shell.h> Shell statistics structure.

Public Members

atomic_t `log_lost_cnt`

Lost log counter.

struct `shell_flags`

#include <shell.h>

Public Members

uint32_t `insert_mode`
Controls insert mode for text introduction.

uint32_t `use_colors`
Controls colored syntax.

uint32_t `echo`
Controls shell echo.

uint32_t `obscure`
If echo on, print asterisk instead

uint32_t `processing`
Shell is executing process function.

uint32_t `mode_delete`
Operation mode of backspace key

uint32_t `history_exit`
Request to exit history mode

uint32_t `last_nl`
Last received new line character

uint32_t `cmd_ctx`
Shell is executing command

uint32_t `print_noinit`
Print request from not initialized shell

uint32_t `panic_mode`
Shell in panic mode

union `shell_internal`
#include <shell.h>

Public Members

uint32_t `value`

struct *shell_flags* `flags`

struct `shell_ctx`
#include <shell.h> Shell instance context.

Public Members

const char *`prompt`

shell current prompt.

enum *shell_state* `state`

Internal module state.

enum *shell_receive_state* `receive_state`

Escape sequence indicator. Currently executed command.

const struct *shell_static_entry* *`selected_cmd`

VT100 color and cursor position, terminal width.

struct shell_vt100_ctx `vt100_ctx`

Callback called from shell thread context when unitialization is completed just before aborting shell thread.

shell_uninit_cb_t `uninit_cb`

When bypass is set, all incoming data is passed to the callback.

uint16_t `cmd_buff_len`

Command length.

uint16_t `cmd_buff_pos`

Command buffer cursor position.

uint16_t `cmd_tmp_buff_len`

Command length in tmp buffer. Command input buffer.

char `cmd_buff`[0]

Command temporary buffer.

char `temp_buff`[0]

Printf buffer size.

volatile union *shell_internal* `internal`

Internal shell data.

struct *k_poll_signal* `signals`[*SHELL_SIGNALS*]

Events that should be used only internally by shell thread. Event for SHELL_SIGNAL_TXDONE is initialized but unused.

struct `shell`

#include <shell.h> Shell instance internals.

Public Members

const char *default_prompt
: shell default prompt.

const struct *shell_transport* *iface
: Transport interface.

struct *shell_ctx* *ctx
: Internal context.

7.26 Storage

7.26.1 Non-Volatile Storage (NVS)

Elements, represented as id-data pairs, are stored in flash using a FIFO-managed circular buffer. The flash area is divided into sectors. Elements are appended to a sector until storage space in the sector is exhausted. Then a new sector in the flash area is prepared for use (erased). Before erasing the sector it is checked that identifier - data pairs exist in the sectors in use, if not the id-data pair is copied.

The id is a 16-bit unsigned number. NVS ensures that for each used id there is at least one id-data pair stored in flash at all time.

NVS allows storage of binary blobs, strings, integers, longs, and any combination of these.

Each element is stored in flash as metadata (8 byte) and data. The metadata is written in a table starting from the end of a nvs sector, the data is written one after the other from the start of the sector. The metadata consists of: id, data offset in sector, data length, part (unused) and a crc.

A write of data to nvs always starts with writing the data, followed by a write of the metadata. Data that is written in flash without metadata is ignored during initialization.

During initialization NVS will verify the data stored in flash, if it encounters an error it will ignore any data with missing/incorrect metadata.

NVS checks the id-data pair before writing data to flash. If the id-data pair is unchanged no write to flash is performed.

To protect the flash area against frequent erases it is important that there is sufficient free space. NVS has a protection mechanism to avoid getting in a endless loop of flash page erases when there is limited free space. When such a loop is detected NVS returns that there is no more space available.

For NVS the file system is declared as:

```
static struct nvs_fs fs = {
.sector_size = NVS_SECTOR_SIZE,
.sector_count = NVS_SECTOR_COUNT,
.offset = NVS_STORAGE_OFFSET,
};
```

where

- `NVS_SECTOR_SIZE` is the sector size, it has to be a multiple of the flash erase page size and a power of 2.
- `NVS_SECTOR_COUNT` is the number of sectors, it is at least 2, one sector is always kept empty to allow copying of existing data.
- `NVS_STORAGE_OFFSET` is the offset of the storage area in flash.

Flash wear

When writing data to flash a study of the flash wear is important. Flash has a limited life which is determined by the number of times flash can be erased. Flash is erased one page at a time and the pagesize is determined by the hardware. As an example a nRF51822 device has a pagesize of 1024 bytes and each page can be erased about 20,000 times.

Calculating expected device lifetime Suppose we use a 4 bytes state variable that is changed every minute and needs to be restored after reboot. NVS has been defined with a sector_size equal to the pagesize (1024 bytes) and 2 sectors have been defined.

Each write of the state variable requires 12 bytes of flash storage: 8 bytes for the metadata and 4 bytes for the data. When storing the data the first sector will be full after 1024/12 = 85.33 minutes. After another 85.33 minutes, the second sector is full. When this happens, because we're using only two sectors, the first sector will be used for storage and will be erased after 171 minutes of system time. With the expected device life of 20,000 writes, with two sectors writing every 171 minutes, the device should last about 171 * 20,000 minutes, or about 6.5 years.

More generally then, with

- `NS` as the number of storage requests per minute,
- `DS` as the data size in bytes,
- `SECTOR_SIZE` in bytes, and
- `PAGE_ERASES` as the number of times the page can be erased,

the expected device life (in minutes) can be calculated as:

```
SECTOR_COUNT * SECTOR_SIZE * PAGE_ERASES / (NS * (DS+8)) minutes
```

From this formula it is also clear what to do in case the expected life is too short: increase `SECTOR_COUNT` or `SECTOR_SIZE`.

Flash write block size migration

It is possible that during a DFU process, the flash driver used by the NVS changes the supported minimal write block size. The NVS in-flash image will stay compatible unless the physical ATE size changes. Especially, migration between 1,2,4,8-bytes write block sizes is allowed.

Sample

A sample of how NVS can be used is supplied in `samples/subsys/nvs`.

Troubleshooting

MPU fault while using NVS, or `-ETIMEDOUT` error returned NVS can use the internal flash of the SoC. While the MPU is enabled, the flash driver requires MPU RWX access to flash memory, configured using `CONFIG_MPU_ALLOW_FLASH_WRITE`. If this option is disabled, the NVS application will get an MPU fault if it references the internal SoC flash and it's the only thread running. In a multi-threaded application, another thread might intercept the fault and the NVS API will return an `-ETIMEDOUT` error.

API Reference

The NVS subsystem APIs are provided by `nvs.h`:

group `nvs_data_structures`

Non-volatile Storage Data Structures.

struct `nvs_fs`

#include <nvs.h> Non-volatile Storage File system structure.

Param offset File system offset in flash

Param ate_wra Allocation table entry write address. Addresses are stored as uint32_t: high 2 bytes correspond to the sector, low 2 bytes are the offset in the sector

Param data_wra Data write address

Param sector_size File system is split into sectors, each sector must be multiple of pagesize

Param sector_count Number of sectors in the file systems

Param ready Flag indicating if the filesystem is initialized

Param nvs_lock Mutex

Param flash_device Flash Device runtime structure

Param flash_parameters Flash memory parameters structure

group `nvs_high_level_api`

Non-volatile Storage APIs.

Functions

int `nvs_init`(struct *nvs_fs* *fs, const char *dev_name)

nvs_init

Initializes a NVS file system in flash.

Parameters

- `fs` – Pointer to file system
- `dev_name` – Pointer to flash device name

Return values

- `0` – Success
- `-ERRNO` – errno code if error

int `nvs_clear`(struct *nvs_fs* *fs)

nvs_clear

Clears the NVS file system from flash.

Parameters

- `fs` – Pointer to file system

Return values

- `0` – Success
- `-ERRNO` – errno code if error

ssize_t `nvs_write`(struct *nvs_fs* *fs, uint16_t id, const void *data, size_t len)

nvs_write

Write an entry to the file system.

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be written
- `data` – Pointer to the data to be written
- `len` – Number of bytes to be written

Returns Number of bytes written. On success, it will be equal to the number of bytes requested to be written. When a rewrite of the same data already stored is attempted, nothing is written to flash, thus 0 is returned. On error, returns negative value of errno.h defined error codes.

int `nvs_delete`(struct *nvs_fs* *fs, uint16_t id)

nvs_delete

Delete an entry from the file system

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be deleted

Return values

- `0` – Success
- `-ERRNO` – errno code if error

ssize_t `nvs_read`(struct *nvs_fs* *fs, uint16_t id, void *data, size_t len)

nvs_read

Read an entry from the file system.

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be read
- `data` – Pointer to data buffer
- `len` – Number of bytes to be read

Returns Number of bytes read. On success, it will be equal to the number of bytes requested to be read. When the return value is larger than the number of bytes requested to read this indicates not all bytes were read, and more data is available. On error, returns negative value of errno.h defined error codes.

ssize_t `nvs_read_hist`(struct *nvs_fs* *fs, uint16_t id, void *data, size_t len, uint16_t cnt)

nvs_read_hist

Read a history entry from the file system.

Parameters

- `fs` – Pointer to file system
- `id` – Id of the entry to be read
- `data` – Pointer to data buffer
- `len` – Number of bytes to be read
- `cnt` – History counter: 0: latest entry, 1: one before latest ...

Returns Number of bytes read. On success, it will be equal to the number of bytes requested to be read. When the return value is larger than the number of bytes requested to read this indicates not all bytes were read, and more data is available. On error, returns negative value of errno.h defined error codes.

ssize_t `nvs_calc_free_space`(struct *nvs_fs* *fs)

nvs_calc_free_space

Calculate the available free space in the file system.

Parameters

- `fs` – Pointer to file system

Returns Number of bytes free. On success, it will be equal to the number of bytes that can still be written to the file system. Calculating the free space is a time consuming operation, especially on spi flash. On error, returns negative value of errno.h defined error codes.

7.26.2 Disk Access

Overview

The disk access API provides access to storage devices.

SD Card support

Zephyr has support for some SD card controllers and support for interfacing SD cards via SPI. These drivers use disk driver interface and a file system can access the SD cards via disk access API. Both standard and high-capacity SD cards are supported.

Note: The system does not support inserting or removing cards while the system is running. The cards must be present at boot and must not be removed. This may be fixed in future releases.

FAT filesystems are not power safe so the filesystem may become corrupted if power is lost or if the card is removed.

SD Card support via SPI Example devicetree fragment below shows how to add SD card node to `spi1` interface. Example uses pin `PA27` for chip select, and runs the SPI bus at 24 MHz once the SD card has been initialized:

```
&spi1 {
        status = "okay";
        cs-gpios = <&porta 27 GPIO_ACTIVE_LOW>;

        sdhc0: sdhc@0 {
                compatible = "zephyr,mmc-spi-slot";
                reg = <0>;
                status = "okay";
                label = "SDHC0";
                spi-max-frequency = <24000000>;
        };
};
```

The SD card will be automatically detected and initialized by the filesystem driver when the board boots.

To read and write files and directories, see the *File Systems* in include/fs.h such as *fs_open()*, *fs_read()*, and *fs_write()*.

Disk Access API Configuration Options

Related configuration options:

- CONFIG_DISK_ACCESS

API Reference

group disk_access_interface

Disk Access APIs.

Functions

int disk_access_init(const char *pdrv)

perform any initialization

This call is made by the consumer before doing any IO calls so that the disk or the backing device can do any initialization.

Parameters

- pdrv – **[in]** Disk name

Returns 0 on success, negative errno code on fail

int disk_access_status(const char *pdrv)

Get the status of disk.

This call is used to get the status of the disk

Parameters

- pdrv – **[in]** Disk name

Returns DISK_STATUS_OK or other DISK_STATUS_*s

int disk_access_read(const char *pdrv, uint8_t *data_buf, uint32_t start_sector, uint32_t num_sector)

read data from disk

Function to read data from disk to a memory buffer.

Parameters

- pdrv – **[in]** Disk name
- data_buf – **[in]** Pointer to the memory buffer to put data.
- start_sector – **[in]** Start disk sector to read from
- num_sector – **[in]** Number of disk sectors to read

Returns 0 on success, negative errno code on fail

int disk_access_write(const char *pdrv, const uint8_t *data_buf, uint32_t start_sector, uint32_t num_sector)

write data to disk

Function write data from memory buffer to disk.

Parameters

- pdrv – **[in]** Disk name
- data_buf – **[in]** Pointer to the memory buffer

- `start_sector` – **[in]** Start disk sector to write to
- `num_sector` – **[in]** Number of disk sectors to write

Returns 0 on success, negative errno code on fail

int `disk_access_ioctl`(const char *pdrv, uint8_t cmd, void *buff)

Get/Configure disk parameters.

Function to get disk parameters and make any special device requests.

Parameters

- `pdrv` – **[in]** Disk name
- `cmd` – **[in]** DISK_IOCTL_* code describing the request
- `buff` – **[in]** Command data buffer

Returns 0 on success, negative errno code on fail

Disk Driver Configuration Options

Related driver configuration options:

- `CONFIG_DISK_DRIVERS`

Disk Driver Interface

group `disk_driver_interface`

Disk Driver Interface.

Defines

`DISK_IOCTL_GET_SECTOR_COUNT`

Possible Cmd Codes for disk_ioctl()

Get the number of sectors in the disk

`DISK_IOCTL_GET_SECTOR_SIZE`

Get the size of a disk SECTOR in bytes

`DISK_IOCTL_RESERVED`

reserved. It used to be DISK_IOCTL_GET_DISK_SIZE

`DISK_IOCTL_GET_ERASE_BLOCK_SZ`

How many sectors constitute a FLASH Erase block

`DISK_IOCTL_CTRL_SYNC`

Commit any cached read/writes to disk

`DISK_STATUS_OK`

Possible return bitmasks for disk_status()

Disk status okay

`DISK_STATUS_UNINIT`
Disk status uninitialized

`DISK_STATUS_NOMEDIA`
Disk status no media

`DISK_STATUS_WR_PROTECT`
Disk status write protected

Functions

int `disk_access_register`(struct *disk_info* *disk)
Register disk.

Parameters

- `disk` – **[in]** Pointer to the disk info structure

Returns 0 on success, negative errno code on fail

int `disk_access_unregister`(struct *disk_info* *disk)
Unregister disk.

Parameters

- `disk` – **[in]** Pointer to the disk info structure

Returns 0 on success, negative errno code on fail

struct `disk_info`
#include <disk.h> Disk info.

Public Members

sys_dnode_t `node`
Internally used list node

char *`name`
Disk name

const struct *disk_operations* *`ops`
Disk operations

const struct *device* *`dev`
Device associated to this disk

struct `disk_operations`
#include <disk.h> Disk operations.

7.26.3 Flash map

The <storage/flash_map.h> API allows accessing information about device flash partitions via *flash_area* structures.

Each struct flash_area describes a flash partition. The API provides access to a "flash map", which contains predefined flash areas accessible via globally unique ID numbers. You can also create flash_area structures at runtime for application-specific purposes.

The flash_area structure contains the name of the flash device the partition is part of; this name can be passed to *device_get_binding()* to get the corresponding *device* structure which can be read and written to using the *flash API*. The flash_area also contains the start offset and size of the partition within the flash memory the device represents.

The flash_map.h API provides functions for operating on a flash_area. The main examples are *flash_area_read()* and *flash_area_write()*. These functions are basically wrappers around the flash API with input parameter range checks. Not all flash APIs have flash_map.h wrappers, but *flash_area_get_device()* allows easily retrieving the struct device from a struct flash_area.

Use *flash_area_open()* to access a struct flash_area. This function takes a flash area ID number and returns a pointer to the flash area structure. The ID number for a flash area can be obtained from a human-readable "label" using *FLASH_AREA_ID*; these labels are obtained from the devicetree as described below.

Relationship with Devicetree

The flash_map.h API uses data generated from the *Devicetree API*, in particular its *Fixed flash partitions*. Zephyr additionally has some partitioning conventions used for *Device Firmware Upgrade* via the MCUboot bootloader, as well as defining partitions usable by *file systems* or other nonvolatile *storage*.

Here is an example devicetree fragment which uses fixed flash partitions for both MCUboot and a storage partition. Some details were left out for clarity.

```
/ {
        soc {
                flashctrl: flash-controller@deadbeef {
                        flash0: flash@0 {
                                compatible = "soc-nv-flash";
                                reg = <0x0 0x100000>;

                                partitions {
                                        compatible = "fixed-partitions";
                                        #address-cells = <0x1>;
                                        #size-cells = <0x1>;

                                        boot_partition: partition@0 {
                                                label = "mcuboot";
                                                reg = <0x0 0x10000>;
                                                read-only;
                                        };
                                        storage_partition: partition@1e000 {
                                                label = "storage";
                                                reg = <0x1e000 0x2000>;
                                        };
                                        slot0_partition: partition@20000 {
                                                label = "image-0";
                                                reg = <0x20000 0x60000>;
                                        };
                                        slot1_partition: partition@80000 {
```

(continues on next page)

(continued from previous page)

```
                                                label = "image-1";
                                                reg = <0x80000 0x60000>;
                                        };
                                        scratch_partition: partition@e0000 {
                                                label = "image-scratch";
                                                reg = <0xe0000 0x20000>;
                                        };
                                };
                        };
                };
        };
};
```

Rule for offsets is that each partition offset shall be expressed in relation to the flash memory beginning address to which the partition belong.

The `boot_partition`, `slot0_partition`, `slot1_partition`, and `scratch_partition` nodes are defined for MCUboot, though not all MCUboot configurations require all of them to be defined. See the MCUboot documentation for more details.

The `storage_partition` node is defined for use by a file system or other nonvolatile storage API.

To get a numeric flash area ID from one of the child nodes of the `partitions` node:

1. take the node's `label` property value
2. lowercase it
3. convert all special characters to underscores (`_`)
4. pass the result **without quotes** to `FLASH_AREA_ID()`

For example, the `flash_area` ID number for `slot0_partition` is `FLASH_AREA_ID(image_0)`.

The same rules apply for other macros which take a "label", such as *`FLASH_AREA_OFFSET`* and *`FLASH_AREA_SIZE`*. For example, `FLASH_AREA_OFFSET(image_0)` would return the start offset for `slot0_partition` within its flash device. This is determined by the node's *reg property*, and in this case is 0x20000.

To get a pointer to the flash area structure and do something with it starting with a devicetree label like `"image-0"`, use something like this:

```
struct flash_area *my_area;
int err = flash_area_open(FLASH_AREA_ID(image_0), &my_area);

if (err != 0) {
     handle_the_error(err);
} else {
     flash_area_read(my_area, ...);
}
```

API Reference

group `flash_area_api`

Abstraction over flash partitions/areas and their drivers.

Defines

SOC_FLASH_0_ID

Provided for compatibility with MCUboot

SPI_FLASH_0_ID

Provided for compatibility with MCUboot

FLASH_AREA_LABEL_EXISTS(label)

FLASH_AREA_LABEL_STR(lbl)

FLASH_AREA_ID(label)

FLASH_AREA_OFFSET(label)

FLASH_AREA_SIZE(label)

Typedefs

typedef void (*flash_area_cb_t)(const struct *flash_area* *fa, void *user_data)

Flash map iteration callback

Param fa flash area

Param user_data User supplied data

Functions

int flash_area_open(uint8_t id, const struct *flash_area* **fa)

Retrieve partitions flash area from the flash_map.

Function Retrieves *flash_area* from flash_map for given partition.

Parameters

- id – **[in]** ID of the flash partition.
- fa – **[out]** Pointer which has to reference *flash_area*. If ID is unknown, it will be NULL on output.

Returns 0 on success, -EACCES if the flash_map is not available , -ENOENT if ID is unknown.

void flash_area_close(const struct *flash_area* *fa)

Close *flash_area*.

Reserved for future usage and external projects compatibility reason. Currently is NOP.

Parameters

- fa – **[in]** Flash area to be closed.

int flash_area_read(const struct *flash_area* *fa, off_t off, void *dst, size_t len)

Read flash area data.

Read data from flash area. Area readout boundaries are asserted before read request. API has the same limitation regard read-block alignment and size as wrapped flash driver.

Parameters

- fa – **[in]** Flash area
- off – **[in]** Offset relative from beginning of flash area to read

- `dst` – **[out]** Buffer to store read data
- `len` – **[in]** Number of bytes to read

Returns 0 on success, negative errno code on fail.

int `flash_area_write`(const struct *flash_area* *fa, off_t off, const void *src, size_t len)

Write data to flash area.

Write data to flash area. Area write boundaries are asserted before write request. API has the same limitation regard write-block alignment and size as wrapped flash driver.

Parameters

- `fa` – **[in]** Flash area
- `off` – **[in]** Offset relative from beginning of flash area to read
- `src` – **[out]** Buffer with data to be written
- `len` – **[in]** Number of bytes to write

Returns 0 on success, negative errno code on fail.

int `flash_area_erase`(const struct *flash_area* *fa, off_t off, size_t len)

Erase flash area.

Erase given flash area range. Area boundaries are asserted before erase request. API has the same limitation regard erase-block alignment and size as wrapped flash driver.

Parameters

- `fa` – **[in]** Flash area
- `off` – **[in]** Offset relative from beginning of flash area.
- `len` – **[in]** Number of bytes to be erase

Returns 0 on success, negative errno code on fail.

uint8_t `flash_area_align`(const struct *flash_area* *fa)

Get write block size of the flash area.

Currently write block size might be treated as read block size, although most of drivers supports unaligned readout.

Parameters

- `fa` – **[in]** Flash area

Returns Alignment restriction for flash writes in [B].

int `flash_area_get_sectors`(int fa_id, uint32_t *count, struct *flash_sector* *sectors)

Retrieve info about sectors within the area.

Parameters

- `fa_id` – **[in]** Given flash area ID
- `sectors` – **[out]** buffer for sectors data
- `count` – **[inout]** On input Capacity of `sectors`, on output number of sectors Retrieved.

Returns 0 on success, negative errno code on fail. Especially returns -ENOMEM if There are too many flash pages on the *flash_area* to fit in the array.

void `flash_area_foreach`(*flash_area_cb_t* user_cb, void *user_data)

Iterate over flash map

Parameters

- `user_cb` – User callback
- `user_data` – User supplied data

int `flash_area_has_driver`(const struct *flash_area* *fa)

Check whether given flash area has supporting flash driver in the system.

Parameters

- `fa` – **[in]** Flash area.

Returns 1 On success. -ENODEV if no driver match.

const struct *device* *`flash_area_get_device`(const struct *flash_area* *fa)

Get driver for given flash area.

Parameters

- `fa` – Flash area.

Returns device driver.

uint8_t `flash_area_erased_val`(const struct *flash_area* *fa)

Get the value expected to be read when accessing any erased flash byte. This API is compatible with the MCUBoot's porting layer.

Parameters

- `fa` – Flash area.

Returns Byte value of erase memory.

struct `flash_area`

#include <flash_map.h> Flash partition.

This structure represents a fixed-size partition on a flash device. Each partition contains one or more flash sectors.

Public Members

uint8_t `fa_id`

ID number

uint8_t `fa_device_id`

Provided for compatibility with MCUboot

off_t `fa_off`

Start offset from the beginning of the flash device

size_t `fa_size`

Total size

const char *`fa_dev_name`

Name of the flash device, suitable for passing to *device_get_binding()*.

struct `flash_sector`

#include <flash_map.h> Structure for transfer flash sector boundaries.

This template is used for presentation of flash memory structure. It consumes much less RAM than *flash_area*

Public Members

off_t `fs_off`

Sector offset from the beginning of the flash device

size_t `fs_size`

Sector size in bytes

7.26.4 Flash Circular Buffer (FCB)

Flash circular buffer provides an abstraction through which you can treat flash like a FIFO. You append entries to the end, and read data from the beginning.

Note: As of Zephyr release 2.1 the *NVS* storage API is recommended over FCB for use as a back-end for the *settings API*.

Description

Entries in the flash contain the length of the entry, the data within the entry, and checksum over the entry contents.

Storage of entries in flash is done in a FIFO fashion. When you request space for the next entry, space is located at the end of the used area. When you start reading, the first entry served is the oldest entry in flash.

Entries can be appended to the end of the area until storage space is exhausted. You have control over what happens next; either erase oldest block of data, thereby freeing up some space, or stop writing new data until existing data has been collected. FCB treats underlying storage as an array of flash sectors; when it erases old data, it does this a sector at a time.

Entries in the flash are checksummed. That is how FCB detects whether writing entry to flash completed ok. It will skip over entries which don't have a valid checksum.

Usage

To add an entry to circular buffer:

- Call *fcb_append* to get the location where data can be written. If this fails due to lack of space, you can call *fcb_rotate* to erase the oldest sector which will make the space. And then call *fcb_append* again.
- Use *flash_area_write* to write entry contents.
- Call *fcb_append_finish* when done. This completes the writing of the entry by calculating the checksum.

To read contents of the circular buffer:

- Call *fcb_walk* with a pointer to your callback function.
- Within callback function copy in data from the entry using *flash_area_read*. You can tell when all data from within a sector has been read by monitoring the returned entry's area pointer. Then you can call *fcb_rotate*, if you're done with that data.

Alternatively:

- Call *fcb_getnext* with 0 in entry offset to get the pointer to the oldest entry.

- Use *flash_area_read* to read entry contents.
- Call *fcb_getnext* with pointer to current entry to get the next one. And so on.

API Reference

The FCB subsystem APIs are provided by `fcb.h`:

Data structures

group `fcb_data_structures`

Defines

`FCB_MAX_LEN`

Max length of element

`FCB_ENTRY_FA_DATA_OFF(entry)`

Helper macro for calculating the data offset related to the fcb *flash_area* start offset.

Parameters

- `entry` – fcb entry structure

struct `fcb_entry`

#include <fcb.h> FCB entry info structure. This data structure describes the element location in the flash.

You would use it to figure out what parameters to pass to *flash_area_read()* to read element contents. Or to *flash_area_write()* when adding a new element. Entry location is pointer to area (within fcb->f_sectors), and offset within that area.

Public Members

struct *flash_sector* *`fe_sector`

Pointer to info about sector where data are placed

uint32_t `fe_elem_off`

Offset from the start of the sector to beginning of element.

uint32_t `fe_data_off`

Offset from the start of the sector to the start of element.

uint16_t `fe_data_len`

Size of data area in fcb entry

struct `fcb_entry_ctx`

#include <fcb.h> Structure for transferring complete information about FCB entry location within flash memory.

Public Members

struct *fcb_entry* `loc`

FCB entry info

const struct *flash_area* *`fap`

Flash area where the entry is placed

struct `fcb`

#include <fcb.h> FCB instance structure.

The following data structure describes the FCB itself. First part should be filled in by the user before calling *fcb_init*. The second part is used by FCB for its internal bookkeeping.

Public Members

uint32_t `f_magic`

Magic value, should not be 0xFFFFFFFF. It is xored with inversion of f_erase_value and placed in the beginning of FCB flash sector. FCB uses this when determining whether sector contains valid data or not. Giving it value of 0xFFFFFFFF means leaving bytes of the filed in "erased" state.

uint8_t `f_version`

Current version number of the data

uint8_t `f_sector_cnt`

Number of elements in sector array

uint8_t `f_scratch_cnt`

Number of sectors to keep empty. This can be used if you need to have scratch space for garbage collecting when FCB fills up.

struct *flash_sector* *`f_sectors`

Array of sectors, must be contiguous

struct *k_mutex* `f_mtx`

Locking for accessing the FCB data, internal state

struct *flash_sector* *`f_oldest`

Pointer to flash sector containing the oldest data, internal state

struct *fcb_entry* `f_active`

internal state

uint16_t `f_active_id`

Flash location where the newest data is, internal state

uint8_t `f_align`

writes to flash have to aligned to this, internal state

const struct *flash_area* *fap

Flash area used by the fcb instance, , internal state. This can be transfer to FCB user

uint8_t f_erase_value

The value flash takes when it is erased. This is read from flash parameters and initialized upon call to fcb_init.

API functions

group fcb_api

Flash Circular Buffer APIs.

Typedefs

typedef int (*fcb_walk_cb)(struct *fcb_entry_ctx* *loc_ctx, void *arg)

FCB Walk callback function type.

Type of function which is expected to be called while walking over fcb entries thanks to a *fcb_walk* call.

Entry data can be read using *flash_area_read()*, using loc_ctx fields as arguments. If cb wants to stop the walk, it should return non-zero value.

Param loc_ctx **[in]** entry location information (full context)

Param arg **[inout]** callback context, transferred from *fcb_walk*.

Return 0 continue walking, non-zero stop walking.

Functions

int fcb_init(int f_area_id, struct *fcb* *fcb)

Initialize FCB instance.

Parameters

- f_area_id – **[in]** ID of flash area where fcb storage resides.
- fcb – **[inout]** FCB instance structure.

Returns 0 on success, non-zero on failure.

int fcb_append(struct *fcb* *fcb, uint16_t len, struct *fcb_entry* *loc)

Appends an entry to circular buffer.

When writing the contents for the entry, use loc->fe_sector and loc->fe_data_off with *flash_area_write()* to fcb *flash_area*. When you're finished, call *fcb_append_finish()* with loc as argument.

Parameters

- fcb – **[in]** FCB instance structure.
- len – **[in]** Length of data which are expected to be written as the entry payload.
- loc – **[out]** entry location information

Returns 0 on success, non-zero on failure.

int `fcb_append_finish`(struct *fcb* *fcb, struct *fcb_entry* *append_loc)

Finishes entry append operation.

Parameters

- `fcb` – **[in]** FCB instance structure.
- `append_loc` – **[in]** entry location information

Returns 0 on success, non-zero on failure.

int `fcb_walk`(struct *fcb* *fcb, struct *flash_sector* *sector, *fcb_walk_cb* cb, void *cb_arg)

Walk over all entries in the FCB sector

Parameters

- `sector` – **[in]** fcb sector to be walked. If null, traverse entire storage.
- `fcb` – **[in]** FCB instance structure.
- `cb` – **[in]** pointer to the function which gets called for every entry. If cb wants to stop the walk, it should return non-zero value.
- `cb_arg` – **[inout]** callback context, transferred to the callback implementation.

Returns 0 on success, negative on failure (or transferred form callback return-value), positive transferred form callback return-value

int `fcb_getnext`(struct *fcb* *fcb, struct *fcb_entry* *loc)

Get next fcb entry location.

Function to obtain fcb entry location in relation to entry pointed by

loc. If loc->fe_sector is set and loc->fe_elem_off is not 0 function fetches next fcb entry location. If loc->fe_sector is NULL function fetches the oldest entry location within FCB storage. loc->fe_sector is set and loc->fe_elem_off is 0 function fetches the first entry location in the fcb sector.

Parameters

- `fcb` – **[in]** FCB instance structure.
- `loc` – **[inout]** entry location information

Returns 0 on success, non-zero on failure.

int `fcb_rotate`(struct *fcb* *fcb)

int `fcb_append_to_scratch`(struct *fcb* *fcb)

int `fcb_free_sector_cnt`(struct *fcb* *fcb)

Get free sector count.

Parameters

- `fcb` – **[in]** FCB instance structure.

Returns Number of free sectors.

int `fcb_is_empty`(struct *fcb* *fcb)

Check whether FCB has any data.

Parameters

- `fcb` – **[in]** FCB instance structure.

Returns Positive value if fcb is empty, otherwise 0.

int `fcb_offset_last_n`(struct *fcb* *fcb, uint8_t entries, struct *fcb_entry* *last_n_entry)

Finds the fcb entry that gives back up to n entries at the end.

Parameters

- `fcb` – **[in]** FCB instance structure.
- `entries` – **[in]** number of fcb entries the user wants to get
- `last_n_entry` – **[out]** last_n_entry the *fcb_entry* to be returned

Returns 0 on there are any fcbs available; -ENOENT otherwise

int `fcb_clear`(struct *fcb* *fcb)

Clear fcb instance storage.

Parameters

- `fcb` – **[in]** FCB instance structure.

Returns 0 on success; non-zero on failure

7.26.5 Stream Flash

The Stream Flash module takes contiguous fragments of a stream of data (e.g. from radio packets), aggregates them into a user-provided buffer, then when the buffer fills (or stream ends) writes it to a raw flash partition. It supports providing the read-back buffer to the client to use in validating the persisted stream content.

One typical use of a stream write operation is when receiving a new firmware image to be used in a DFU operation.

There are several reasons why one might want to use buffered writes instead of writing the data directly as it is made available. Some devices have hardware limitations which does not allow flash writes to be performed in parallell with other operations, such as radio RX and TX. Also, fewer write operations result in faster response times seen from the application.

Persistent stream write progress

Some stream write operations, such as DFU operations, may run for a long time. When performing such long running operations it can be useful to be able to save the stream write progress to persistent storage so that the operation can resume at the same point after an unexpected interruption.

The Stream Flash module offers an API for loading, saving and clearing stream write progress to persistent storage using the *Settings* module. The API can be enabled using `CONFIG_STREAM_FLASH_PROGRESS`.

API Reference

group `stream_flash`

Abstraction over stream writes to flash.

Typedefs

typedef int (*`stream_flash_callback_t`)(uint8_t *buf, size_t len, size_t offset)

Signature for callback invoked after flash write completes.

Functions of this type are invoked with a buffer containing data read back from the flash after a flash write has completed. This enables verifying that the data has been correctly stored

(for instance by using a SHA function). The write buffer 'buf' provided in stream_flash_init is used as a read buffer for this purpose.

Param buf Pointer to the data read.

Param len The length of the data read.

Param offset The offset the data was read from.

Functions

int `stream_flash_init`(struct *stream_flash_ctx* *ctx, const struct *device* *fdev, uint8_t *buf, size_t buf_len, size_t offset, size_t size, *stream_flash_callback_t* cb)

Initialize context needed for stream writes to flash.

Parameters

- `ctx` – context to be initialized
- `fdev` – Flash device to operate on
- `buf` – Write buffer
- `buf_len` – Length of write buffer. Can not be larger than the page size. Must be multiple of the flash device write-block-size.
- `offset` – Offset within flash device to start writing to
- `size` – Number of bytes available for performing buffered write. If this is '0', the size will be set to the total size of the flash device minus the offset.
- `cb` – Callback to be invoked on completed flash write operations.

Returns non-negative on success, negative errno code on fail

size_t `stream_flash_bytes_written`(struct *stream_flash_ctx* *ctx)

Read number of bytes written to the flash.

Note: api-tags: pre-kernel-ok isr-ok

Parameters

- `ctx` – context

Returns Number of payload bytes written to flash.

int `stream_flash_buffered_write`(struct *stream_flash_ctx* *ctx, const uint8_t *data, size_t len, bool flush)

Process input buffers to be written to flash device in single blocks. Will store remainder between calls.

A final call to this function with flush set to true will write out the remaining block buffer to flash.

Parameters

- `ctx` – context
- `data` – data to write
- `len` – Number of bytes to write

- `flush` – when true this forces any buffered data to be written to flash A flush write should be the last write operation in a sequence of write operations for given context (although this is not mandatory if the total data size is a multiple of the buffer size).

Returns non-negative on success, negative errno code on fail

int `stream_flash_erase_page`(struct *stream_flash_ctx* *ctx, off_t off)

Erase the flash page to which a given offset belongs.

This function erases a flash page to which an offset belongs if this page is not the page previously erased by the provided ctx (ctx->last_erased_page_start_offset).

Parameters

- `ctx` – context
- `off` – offset from the base address of the flash device

Returns non-negative on success, negative errno code on fail

int `stream_flash_progress_load`(struct *stream_flash_ctx* *ctx, const char *settings_key)

Load persistent stream write progress stored with key `settings_key` .

This function should be called directly after *stream_flash_init* to load previous stream write progress before writing any data. If the loaded progress has fewer bytes written than `ctx` then it will be ignored.

Parameters

- `ctx` – context
- `settings_key` – key to use with the settings module for loading the stream write progress

Returns non-negative on success, negative errno code on fail

int `stream_flash_progress_save`(struct *stream_flash_ctx* *ctx, const char *settings_key)

Save persistent stream write progress using key `settings_key` .

Parameters

- `ctx` – context
- `settings_key` – key to use with the settings module for storing the stream write progress

Returns non-negative on success, negative errno code on fail

int `stream_flash_progress_clear`(struct *stream_flash_ctx* *ctx, const char *settings_key)

Clear persistent stream write progress stored with key `settings_key` .

Parameters

- `ctx` – context
- `settings_key` – key previously used for storing the stream write progress

Returns non-negative on success, negative errno code on fail

struct `stream_flash_ctx`

#include <stream_flash.h> Structure for stream flash context.

Users should treat these structures as opaque values and only interact with them through the below API.

7.27 Task Watchdog

7.27.1 Overview

Many microcontrollers feature a hardware watchdog timer peripheral. Its purpose is to trigger an action (usually a system reset) in case of severe software malfunctions. Once initialized, the watchdog timer has to be restarted ("fed") in regular intervals to prevent it from timing out. If the software got stuck and does not manage to feed the watchdog anymore, the corrective action is triggered to bring the system back to normal operation.

In real-time operating systems with multiple tasks running in parallel, a single watchdog instance may not be sufficient anymore, as it can be used for only one task. This software watchdog based on kernel timers provides a method to supervise multiple threads or tasks (called watchdog channels).

An existing hardware watchdog can be used as an optional fallback if the task watchdog itself or the scheduler has a malfunction.

The task watchdog uses a kernel timer as its backend. If configured properly, the timer ISR is never actually called during normal operation, as the timer is continuously updated in the feed calls.

It's currently not possible to have multiple instances of task watchdogs. Instead, the task watchdog API can be accessed globally to add or delete new channels without passing around a context or device pointer in the firmware.

The maximum number of channels is predefined via Kconfig and should be adjusted to match exactly the number of channels required by the application.

7.27.2 Configuration Options

Related configuration options can be found under subsys/task_wdt/Kconfig.

- `CONFIG_TASK_WDT`
- `CONFIG_TASK_WDT_CHANNELS`
- `CONFIG_TASK_WDT_HW_FALLBACK`
- `CONFIG_TASK_WDT_MIN_TIMEOUT`
- `CONFIG_TASK_WDT_HW_FALLBACK_DELAY`

7.27.3 API Reference

group `task_wdt_api`

Task Watchdog APIs.

Typedefs

typedef void (*`task_wdt_callback_t`)(int channel_id, void *user_data)

Task watchdog callback.

Functions

int `task_wdt_init`(const struct *device* *hw_wdt)

Initialize task watchdog.

This function sets up necessary kernel timers and the hardware watchdog (if desired as fallback). It has to be called before *task_wdt_add()* and *task_wdt_feed()*.

Parameters

- `hw_wdt` – Pointer to the hardware watchdog device used as fallback. Pass NULL if no hardware watchdog fallback is desired.

Return values

- `0` – If successful.
- `-ENOTSUP` – If assigning a hardware watchdog is not supported.

int `task_wdt_add`(uint32_t reload_period, *task_wdt_callback_t* callback, void *user_data)

Install new timeout.

Adds a new timeout to the list of task watchdog channels.

Parameters

- `reload_period` – Period in milliseconds used to reset the timeout
- `callback` – Function to be called when watchdog timer expired. Pass NULL to use system reset handler.
- `user_data` – User data to associate with the watchdog channel.

Return values

- `channel_id` – If successful, a non-negative value indicating the index of the channel to which the timeout was assigned. This ID is supposed to be used as the parameter in calls to *task_wdt_feed()*.
- `-EINVAL` – If the reload_period is invalid.
- `-ENOMEM` – If no more timeouts can be installed.

int `task_wdt_delete`(int channel_id)

Delete task watchdog channel.

Deletes the specified channel from the list of task watchdog channels. The channel is now available again for other tasks via *task_wdt_add()* function.

Parameters

- `channel_id` – Index of the channel as returned by *task_wdt_add()*.

Return values

- `0` – If successful.
- `-EINVAL` – If there is no installed timeout for supplied channel.

int `task_wdt_feed`(int channel_id)

Feed specified watchdog channel.

This function loops through all installed task watchdogs and updates the internal kernel timer used as for the software watchdog with the next due timeout.

Parameters

- `channel_id` – Index of the fed channel as returned by *task_wdt_add()*.

Return values

- `0` – If successful.
- `-EINVAL` – If there is no installed timeout for supplied channel.

7.28 Time Utilities

7.28.1 Overview

Uptime in Zephyr is based on the a tick counter. With the default `CONFIG_TICKLESS_KERNEL` this counter advances at a nominally constant rate from zero at the instant the system started. The POSIX equivalent to this counter is something like `CLOCK_MONOTONIC` or, in Linux, `CLOCK_MONOTONIC_RAW`. *k_uptime_get()* provides a millisecond representation of this time.

Applications often need to correlate the Zephyr internal time with external time scales used in daily life, such as local time or Coordinated Universal Time. These systems interpret time in different ways and may have discontinuities due to leap seconds and local time offsets like daylight saving time.

Because of these discontinuities, as well as significant inaccuracies in the clocks underlying the cycle counter, the offset between time estimated from the Zephyr clock and the actual time in a "real" civil time scale is not constant and can vary widely over the runtime of a Zephyr application.

The time utilities API supports:

- *converting between time representations*
- *synchronizing and aligning time scales*

For terminology and concepts that support these functions see *Concepts Underlying Time Support in Zephyr*.

7.28.2 Time Utility APIs

Representation Transformation

Time scale instants can be represented in multiple ways including:

- Seconds since an epoch. POSIX representations of time in this form include `time_t` and `struct timespec`, which are generally interpreted as a representation of "UNIX Time".
- Calendar time as a year, month, day, hour, minutes, and seconds relative to an epoch. POSIX representations of time in this form include `struct tm`.

Keep in mind that these are simply time representations that must be interpreted relative to a time scale which may be local time, UTC, or some other continuous or discontinuous scale.

Some necessary transformations are available in standard C library routines. For example, `time_t` measuring seconds since the POSIX EPOCH is converted to `struct tm` representing calendar time with gmtime(). Sub-second timestamps like `struct timespec` can also use this to produce the calendar time representation and deal with sub-second offsets separately.

The inverse transformation is not standardized: APIs like `mktime()` expect information about time zones. Zephyr provides this transformation with *timeutil_timegm()* and *timeutil_timegm64()*.

group `timeutil_repr_apis`

Functions

int64_t `timeutil_timegm64`(const struct *tm* *tm)

Convert broken-down time to a POSIX epoch offset in seconds.

See also:

http://man7.org/linux/man-pages/man3/timegm.3.html

Parameters

- `tm` – pointer to broken down time.

Returns the corresponding time in the POSIX epoch time scale.

time_t `timeutil_timegm`(const struct *tm* *tm)

Convert broken-down time to a POSIX epoch offset in seconds.

See also:

http://man7.org/linux/man-pages/man3/timegm.3.html

Parameters

- `tm` – pointer to broken down time.

Returns the corresponding time in the POSIX epoch time scale. If the time cannot be represented then (time_t)-1 is returned and `errno` is set to `ERANGE`.

Time Scale Synchronization

There are several factors that affect synchronizing time scales:

- The rate of discrete instant representation change. For example Zephyr uptime is tracked in ticks which advance at events that nominally occur at `CONFIG_SYS_CLOCK_TICKS_PER_SEC` Hertz, while an external time source may provide data in whole or fractional seconds (e.g. microseconds).
- The absolute offset required to align the two scales at a single instant.
- The relative error between observable instants in each scale, required to align multiple instants consistently. For example a reference clock that's conditioned by a 1-pulse-per-second GPS signal will be much more accurate than a Zephyr system clock driven by a RC oscillator with a +/- 250 ppm error.

Synchronization or alignment between time scales is done with a multi-step process:

- An instant in a time scale is represented by an (unsigned) 64-bit integer, assumed to advance at a fixed nominal rate.
- *timeutil_sync_config* records the nominal rates of a reference time scale/source (e.g. TAI) and a local time source (e.g. *k_uptime_ticks()*).
- *timeutil_sync_instant* records the representation of a single instant in both the reference and local time scales.
- *timeutil_sync_state* provides storage for an initial instant, a recently received second observation, and a skew that can adjust for relative errors in the actual rate of each time scale.
- *timeutil_sync_ref_from_local()* and *timeutil_sync_local_from_ref()* convert instants in one time scale to another taking into account skew that can be estimated from the two instances stored in the state structure by *timeutil_sync_estimate_skew()*.

group `timeutil_sync_apis`

Functions

int `timeutil_sync_state_update`(struct *timeutil_sync_state* *tsp, const struct *timeutil_sync_instant* *inst)

Record a new instant in the time synchronization state.

Note that this updates only the latest persisted instant. The skew is not adjusted automatically.

Parameters

- `tsp` – pointer to a *timeutil_sync_state* object.
- `inst` – the new instant to be recorded. This becomes the base instant if there is no base instant, otherwise the value must be strictly after the base instant in both the reference and local time scales.

Return values

- `0` – if installation succeeded in providing a new base
- `1` – if installation provided a new latest instant
- `-EINVAL` – if the new instant is not compatible with the base instant

int `timeutil_sync_state_set_skew`(struct *timeutil_sync_state* *tsp, float skew, const struct *timeutil_sync_instant* *base)

Update the state with a new skew and possibly base value.

Set the skew from a value retrieved from persistent storage, or calculated based on recent skew estimations including from *timeutil_sync_estimate_skew()*.

Optionally update the base timestamp. If the base is replaced the latest instant will be cleared until *timeutil_sync_state_update()* is invoked.

Parameters

- `tsp` – pointer to a time synchronization state.
- `skew` – the skew to be used. The value must be positive and shouldn't be too far away from 1.
- `base` – optional new base to be set. If provided this becomes the base timestamp that will be used along with skew to convert between reference and local timescale instants. Setting the base clears the captured latest value.

Returns 0 if skew was updated

Returns -EINVAL if skew was not valid

float `timeutil_sync_estimate_skew`(const struct *timeutil_sync_state* *tsp)

Estimate the skew based on current state.

Using the base and latest syncpoints from the state determine the skew of the local clock relative to the reference clock. See *timeutil_sync_state::skew*.

Parameters

- `tsp` – pointer to a time synchronization state. The base and latest syncpoints must be present and the latest syncpoint must be after the base point in the local time scale.

Returns the estimated skew, or zero if skew could not be estimated.

int `timeutil_sync_ref_from_local`(const struct *timeutil_sync_state* *tsp, uint64_t local, uint64_t *refp)

Interpolate a reference timescale instant from a local instant.

Parameters

- `tsp` – pointer to a time synchronization state. This must have a base and a skew installed.
- `local` – an instant measured in the local timescale. This may be before or after the base instant.
- `refp` – where the corresponding instant in the reference timescale should be stored. A negative interpolated reference time produces an error. If interpolation fails the referenced object is not modified.

Return values

- `0` – if interpolated using a skew of 1
- `1` – if interpolated using a skew not equal to 1
- `-EINVAL` –
  - the times synchronization state is not adequately initialized
  - `refp` is null
- `-ERANGE` – the interpolated reference time would be negative

int `timeutil_sync_local_from_ref`(const struct *timeutil_sync_state* *tsp, uint64_t ref, int64_t *localp)

Interpolate a local timescale instant from a reference instant.

Parameters

- `tsp` – pointer to a time synchronization state. This must have a base and a skew installed.
- `ref` – an instant measured in the reference timescale. This may be before or after the base instant.
- `localp` – where the corresponding instant in the local timescale should be stored. An interpolated value before local time 0 is provided without error. If interpolation fails the referenced object is not modified.

Return values

- `0` – if successful with a skew of 1
- `1` – if successful with a skew not equal to 1
- `-EINVAL` –
  - the time synchronization state is not adequately initialized
  - `refp` is null

int32_t `timeutil_sync_skew_to_ppb`(float skew)

Convert from a skew to an error in parts-per-billion.

A skew of 1.0 has zero error. A skew less than 1 has a positive error (clock is faster than it should be). A skew greater than one has a negative error (clock is slower than it should be).

Note that due to the limited precision of `float` compared with `double` the smallest error that can be represented is about 120 ppb. A "precise" time source may have error on the order of 2000 ppb.

A skew greater than 3.14748 may underflow the 32-bit representation; this represents a clock running at less than 1/3 its nominal rate.

Returns skew error represented as parts-per-billion, or INT32_MIN if the skew cannot be represented in the return type.

struct `timeutil_sync_config`

#include <timeutil.h> Immutable state for synchronizing two clocks.

Values required to convert durations between two time scales.

Note: The accuracy of the translation and calculated skew between sources depends on the resolution of these frequencies. A reference frequency with microsecond or nanosecond resolution would produce the most accurate tracking when the local reference is the Zephyr

tick counter. A reference source like an RTC chip with 1 Hz resolution requires a much larger interval between sampled instants to detect relative clock drift.

Public Members

uint32_t `ref_Hz`

The nominal instance counter rate in Hz.

This value is assumed to be precise, but may drift depending on the reference clock source.

The value must be positive.

uint32_t `local_Hz`

The nominal local counter rate in Hz.

This value is assumed to be inaccurate but reasonably stable. For a local clock driven by a crystal oscillator an error of 25 ppm is common; for an RC oscillator larger errors should be expected. The timeutil_sync infrastructure can calculate the skew between the local and reference clocks and apply it when converting between time scales.

The value must be positive.

struct `timeutil_sync_instant`

#include <timeutil.h> Representation of an instant in two time scales.

Capturing the same instant in two time scales provides a registration point that can be used to convert between those time scales.

Public Members

uint64_t `ref`

An instant in the reference time scale.

This must never be zero in an initialized *timeutil_sync_instant* object.

uint64_t `local`

The corresponding instance in the local time scale.

This may be zero in a valid *timeutil_sync_instant* object.

struct `timeutil_sync_state`

#include <timeutil.h> State required to convert instants between time scales.

This state in conjunction with functions that manipulate it capture the offset information necessary to convert between two timescales along with information that corrects for skew due to inaccuracies in clock rates.

State objects should be zero-initialized before use.

Public Members

const struct *timeutil_sync_config* `*cfg`

Pointer to reference and local rate information.

struct *timeutil_sync_instant* `base`

The base instant in both time scales.

struct *timeutil_sync_instant* `latest`

The most recent instant in both time scales.

This is captured here to provide data for skew calculation.

float `skew`

The scale factor used to correct for clock skew.

The nominal rate for the local counter is assumed to be inaccurate but stable, i.e. it will generally be some parts-per-million faster or slower than specified.

A duration in observed local clock ticks must be multiplied by this value to produce a duration in ticks of a clock operating at the nominal local rate.

A zero value indicates that the skew has not been initialized. If the value is zero when *base* is initialized the skew will be set to 1. Otherwise the skew is assigned through *timeutil_sync_state_set_skew()*.

7.28.3 Concepts Underlying Time Support in Zephyr

Terms from ISO/TC 154/WG 5 N0038 (ISO/WD 8601-1) and elsewhere:

- A *time axis* is a representation of time as an ordered sequence of instants.
- A *time scale* is a way of representing an instant relative to an origin that serves as the epoch.
- A time scale is *monotonic* (increasing) if the representation of successive time instants never decreases in value.
- A time scale is *continuous* if the representation has no abrupt changes in value, e.g. jumping forward or back when going between successive instants.
- Civil time generally refers to time scales that legally defined by civil authorities, like local governments, often to align local midnight to solar time.

Relevant Time Scales

International Atomic Time (TAI) is a time scale based on averaging clocks that count in SI seconds. TAI is a montonic and continuous time scale.

Universal Time (UT) is a time scale based on Earth's rotation. UT is a discontinuous time scale as it requires occasional adjustments (leap seconds) to maintain alignment to changes in Earth's rotation. Thus the difference between TAI and UT varies over time. There are several variants of UT, with UTC being the most common.

UT times are independent of location. UT is the basis for Standard Time (or "local time") which is the time at a particular location. Standard time has a fixed offset from UT at any given instant, primarily influenced by longitude, but the offset may be adjusted ("daylight saving time") to align standard time to the local solar time. In a sense local time is "more discontinuous" than UT.

POSIX Time is a time scale that counts seconds since the "POSIX epoch" at 1970-01-01T00:00:00Z (i.e. the start of 1970 UTC). UNIX Time is an extension of POSIX time using negative values to represent times before the POSIX epoch. Both of these scales assume that every day has exactly 86400 seconds. In normal use instants in these scales correspond to times in the UTC scale, so they inherit the discontinuity.

The continuous analogue is UNIX Leap Time which is UNIX time plus all leap-second corrections added after the POSIX epoch (when TAI-UTC was 8 s).

Example of Time Scale Differences A positive leap second was introduced at the end of 2016, increasing the difference between TAI and UTC from 36 seconds to 37 seconds. There was no leap second introduced at the end of 1999, when the difference between TAI and UTC was only 32 seconds. The following table shows relevant civil and epoch times in several scales:

UTC Date	UNIX time	TAI Date	TAI-UTC	UNIX Leap Time
1970-01-01T00:00:00Z	0	1970-01-01T00:00:08	+8	0
1999-12-31T23:59:28Z	946684768	2000-01-01T00:00:00	+32	946684792
1999-12-31T23:59:59Z	946684799	2000-01-01T00:00:31	+32	946684823
2000-01-01T00:00:00Z	946684800	2000-01-01T00:00:32	+32	946684824
2016-12-31T23:59:59Z	1483228799	2017-01-01T00:00:35	+36	1483228827
2016-12-31T23:59:60Z	undefined	2017-01-01T00:00:36	+36	1483228828
2017-01-01T00:00:00Z	1483228800	2017-01-01T00:00:37	+37	1483228829

Functional Requirements The Zephyr tick counter has no concept of leap seconds or standard time offsets and is a continuous time scale. However it can be relatively inaccurate, with drifts as much as three minutes per hour (assuming an RC timer with 5% tolerance).

There are two stages required to support conversion between Zephyr time and common human time scales:

- Translation between the continuous but inaccurate Zephyr time scale and an accurate external stable time scale;
- Translation between the stable time scale and the (possibly discontinuous) civil time scale.

The API around *timeutil_sync_state_update()* supports the first step of converting between continuous time scales.

The second step requires external information including schedules of leap seconds and local time offset changes. This may be best provided by an external library, and is not currently part of the time utility APIs.

Selecting an External Source and Time Scale If an application requires civil time accuracy within several seconds then UTC could be used as the stable time source. However, if the external source adjusts to a leap second there will be a discontinuity: the elapsed time between two observations taken at 1 Hz is not equal to the numeric difference between their timestamps.

For precise activities a continuous scale that is independent of local and solar adjustments simplifies things considerably. Suitable continuous scales include:

- GPS time: epoch of 1980-01-06T00:00:00Z, continuous following TAI with an offset of TAI-GPS=19 s.
- Bluetooth mesh time: epoch of 2000-01-01T00:00:00Z, continuous following TAI with an offset of -32.
- UNIX Leap Time: epoch of 1970-01-01T00:00:00Z, continuous following TAI with an offset of -8.

Because C and Zephyr library functions support conversion between integral and calendar time representations using the UNIX epoch, UNIX Leap Time is an ideal choice for the external time scale.

The mechanism used to populate synchronization points is not relevant: it may involve reading from a local high-precision RTC peripheral, exchanging packets over a network using a protocol like NTP or PTP, or processing NMEA messages received a GPS with or without a 1pps signal.

7.29 USB device support

7.29.1 USB device controller driver API

The USB Device Controller Driver Layer implements the low level control routines to deal directly with the hardware. All device controller drivers should implement the APIs described in include/drivers/usb/usb_dc.h. This allows the integration of new USB device controllers to be done without changing the upper layers. With this API it is not possible to support more than one controller instance at runtime.

API reference

group `_usb_device_controller_api`

USB Device Controller API.

Typedefs

typedef void (*`usb_dc_ep_callback`)(uint8_t ep, enum *usb_dc_ep_cb_status_code* cb_status)

Callback function signature for the USB Endpoint status

typedef void (*`usb_dc_status_callback`)(enum *usb_dc_status_code* cb_status, const uint8_t *param)

Callback function signature for the device

Enums

enum `usb_dc_status_code`

USB Driver Status Codes.

Status codes reported by the registered device status callback.

Values:

enumerator `USB_DC_ERROR`

USB error reported by the controller

enumerator `USB_DC_RESET`

USB reset

enumerator `USB_DC_CONNECTED`

USB connection established, hardware enumeration is completed

enumerator `USB_DC_CONFIGURED`

USB configuration done

enumerator `USB_DC_DISCONNECTED`

USB connection lost

enumerator `USB_DC_SUSPEND`

USB connection suspended by the HOST

enumerator USB_DC_RESUME
USB connection resumed by the HOST

enumerator USB_DC_INTERFACE
USB interface selected

enumerator USB_DC_SET_HALT
Set Feature ENDPOINT_HALT received

enumerator USB_DC_CLEAR_HALT
Clear Feature ENDPOINT_HALT received

enumerator USB_DC_SOF
Start of Frame received

enumerator USB_DC_UNKNOWN
Initial USB connection status

enum usb_dc_ep_cb_status_code
USB Endpoint Callback Status Codes.
Status Codes reported by the registered endpoint callback.
Values:

enumerator USB_DC_EP_SETUP
SETUP received

enumerator USB_DC_EP_DATA_OUT
Out transaction on this EP, data is available for read

enumerator USB_DC_EP_DATA_IN
In transaction done on this EP

enum usb_dc_ep_transfer_type
USB Endpoint Transfer Type.
Values:

enumerator USB_DC_EP_CONTROL = 0
Control type endpoint

enumerator USB_DC_EP_ISOCHRONOUS
Isochronous type endpoint

enumerator USB_DC_EP_BULK
Bulk type endpoint

enumerator USB_DC_EP_INTERRUPT
Interrupt type endpoint

enum usb_dc_ep_synchronozation_type

USB Endpoint Synchronization Type.

Note: Valid only for Isochronous Endpoints

Values:

enumerator USB_DC_EP_NO_SYNCHRONIZATION = (0U << 2U)

No Synchronization

enumerator USB_DC_EP_ASYNCHRONOUS = (1U << 2U)

Asynchronous

enumerator USB_DC_EP_ADAPTIVE = (2U << 2U)

Adaptive

enumerator USB_DC_EP_SYNCHRONOUS = (3U << 2U)

Synchronous

Functions

int usb_dc_attach(void)

Attach USB for device connection.

Function to attach USB for device connection. Upon success, the USB PLL is enabled, and the USB device is now capable of transmitting and receiving on the USB bus and of generating interrupts.

Returns 0 on success, negative errno code on fail.

int usb_dc_detach(void)

Detach the USB device.

Function to detach the USB device. Upon success, the USB hardware PLL is powered down and USB communication is disabled.

Returns 0 on success, negative errno code on fail.

int usb_dc_reset(void)

Reset the USB device.

This function returns the USB device and firmware back to it's initial state. N.B. the USB PLL is handled by the usb_detach function

Returns 0 on success, negative errno code on fail.

int usb_dc_set_address(const uint8_t addr)

Set USB device address.

Parameters

- addr – **[in]** Device address

Returns 0 on success, negative errno code on fail.

void `usb_dc_set_status_callback`(const *usb_dc_status_callback* cb)

Set USB device controller status callback.

Function to set USB device controller status callback. The registered callback is used to report changes in the status of the device controller. The status code are described by the usb_dc_status_code enumeration.

Parameters

- `cb` – **[in]** Callback function

int `usb_dc_ep_check_cap`(const struct *usb_dc_ep_cfg_data* *const cfg)

check endpoint capabilities

Function to check capabilities of an endpoint. *usb_dc_ep_cfg_data* structure provides the endpoint configuration parameters: endpoint address, endpoint maximum packet size and endpoint type. The driver should check endpoint capabilities and return 0 if the endpoint configuration is possible.

Parameters

- `cfg` – **[in]** Endpoint config

Returns 0 on success, negative errno code on fail.

int `usb_dc_ep_configure`(const struct *usb_dc_ep_cfg_data* *const cfg)

Configure endpoint.

Function to configure an endpoint. *usb_dc_ep_cfg_data* structure provides the endpoint configuration parameters: endpoint address, endpoint maximum packet size and endpoint type.

Parameters

- `cfg` – **[in]** Endpoint config

Returns 0 on success, negative errno code on fail.

int `usb_dc_ep_set_stall`(const uint8_t ep)

Set stall condition for the selected endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

int `usb_dc_ep_clear_stall`(const uint8_t ep)

Clear stall condition for the selected endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

int `usb_dc_ep_is_stalled`(const uint8_t ep, uint8_t *const stalled)

Check if the selected endpoint is stalled.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `stalled` – **[out]** Endpoint stall status

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_halt(const uint8_t ep)

Halt the selected endpoint.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_enable(const uint8_t ep)

Enable the selected endpoint.

Function to enable the selected endpoint. Upon success interrupts are enabled for the corresponding endpoint and the endpoint is ready for transmitting/receiving data.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_disable(const uint8_t ep)

Disable the selected endpoint.

Function to disable the selected endpoint. Upon success interrupts are disabled for the corresponding endpoint and the endpoint is no longer able for transmitting/receiving data.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_flush(const uint8_t ep)

Flush the selected endpoint.

This function flushes the FIFOs for the selected endpoint.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_write(const uint8_t ep, const uint8_t *const data, const uint32_t data_len, uint32_t *const ret_bytes)

Write data to the specified endpoint.

This function is called to write data to the specified endpoint. The supplied usb_ep_callback function will be called when data is transmitted out.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- data – **[in]** Pointer to data to write
- data_len – **[in]** Length of the data requested to write. This may be zero for a zero length status packet.
- ret_bytes – **[out]** Bytes scheduled for transmission. This value may be NULL if the application expects all bytes to be written

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_read(const uint8_t ep, uint8_t *const data, const uint32_t max_data_len, uint32_t *const read_bytes)

Read data from the specified endpoint.

This function is called by the endpoint handler function, after an OUT interrupt has been received for that EP. The application must only call this function through the supplied usb_ep_callback function. This function clears the ENDPOINT NAK, if all data in the endpoint FIFO has been read, so as to accept more data from host.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- data – **[in]** Pointer to data buffer to write to
- max_data_len – **[in]** Max length of data to read
- read_bytes – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read should be returned.

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_set_callback(const uint8_t ep, const *usb_dc_ep_callback* cb)

Set callback function for the specified endpoint.

Function to set callback function for notification of data received and available to application or transmit done on the selected endpoint, NULL if callback not required by application code. The callback status code is described by usb_dc_ep_cb_status_code.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- cb – **[in]** Callback function

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_read_wait(uint8_t ep, uint8_t *data, uint32_t max_data_len, uint32_t *read_bytes)

Read data from the specified endpoint.

This is similar to usb_dc_ep_read, the difference being that, it doesn't clear the endpoint NAKs so that the consumer is not bogged down by further upcalls till he is done with the processing of the data. The caller should reactivate ep by invoking *usb_dc_ep_read_continue()* do so.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- data – **[in]** Pointer to data buffer to write to
- max_data_len – **[in]** Max length of data to read
- read_bytes – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read should be returned.

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_read_continue(uint8_t ep)

Continue reading data from the endpoint.

Clear the endpoint NAK and enable the endpoint to accept more data from the host. Usually called after *usb_dc_ep_read_wait()* when the consumer is fine to accept more data. Thus these calls together act as a flow control mechanism.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

int usb_dc_ep_mps(uint8_t ep)

Get endpoint max packet size.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns Enpoint max packet size (mps)

int usb_dc_wakeup_request(void)

Start the host wake up procedure.

Function to wake up the host if it's currently in sleep mode.

Returns 0 on success, negative errno code on fail.

struct usb_dc_ep_cfg_data

#include <usb_dc.h> USB Endpoint Configuration.

Structure containing the USB endpoint configuration.

Public Members

uint8_t ep_addr

The number associated with the EP in the device configuration structure IN EP = 0x80 | <endpoint number> OUT EP = 0x00 | <endpoint number>

uint16_t ep_mps

Endpoint max packet size

enum *usb_dc_ep_transfer_type* ep_type

Endpoint Transfer Type. May be Bulk, Interrupt, Control or Isochronous

7.29.2 USB device stack

The USB device stack is a hardware independent interface between USB device controller driver and USB device class drivers or customer applications. It is a port of the LPCUSB device stack and has been modified and expanded over time. It provides the following functionalities:

- Uses the APIs provided by the device controller drivers to interact with the USB device controller.
- Responds to standard device requests and returns standard descriptors, essentially handling 'Chapter 9' processing, specifically the standard device requests in table 9-3 from the universal serial bus specification revision 2.0.
- Provides a programming interface to be used by USB device classes or customer applications. The APIs is described in include/usb/usb_device.h

The device stack has few limitations with which it is not possible to support more than one controller instance at runtime, and only one USB device configuration is supported.

Supported USB classes:

- USB Audio (experimental)

- USB CDC ACM
- USB CDC ECM
- USB CDC EEM
- RNDIS
- USB MSC
- USB DFU
- Bluetooth HCI over USB
- USB HID class

List of samples for different purposes. CDC ACM and HID samples have configuration overlays for composite configuration.

Implementing a non-standard USB class

The configuration of USB Device is done in the stack layer.

The following structures and callbacks need to be defined:

- Part of USB Descriptor table
- USB Endpoint configuration table
- USB Device configuration structure
- Endpoint callbacks
- Optionally class, vendor and custom handlers

For example, for the USB loopback application:

```
struct usb_loopback_config {
        struct usb_if_descriptor if0;
        struct usb_ep_descriptor if0_out_ep;
        struct usb_ep_descriptor if0_in_ep;
} __packed;

USBD_CLASS_DESCR_DEFINE(primary, 0) struct usb_loopback_config loopback_cfg = {
        /* Interface descriptor 0 */
        .if0 = {
                .bLength = sizeof(struct usb_if_descriptor),
                .bDescriptorType = USB_DESC_INTERFACE,
                .bInterfaceNumber = 0,
                .bAlternateSetting = 0,
                .bNumEndpoints = 2,
                .bInterfaceClass = USB_BCC_VENDOR,
                .bInterfaceSubClass = 0,
                .bInterfaceProtocol = 0,
                .iInterface = 0,
        },

        /* Data Endpoint OUT */
        .if0_out_ep = {
                .bLength = sizeof(struct usb_ep_descriptor),
                .bDescriptorType = USB_DESC_ENDPOINT,
                .bEndpointAddress = LOOPBACK_OUT_EP_ADDR,
                .bmAttributes = USB_DC_EP_BULK,
                .wMaxPacketSize = sys_cpu_to_le16(CONFIG_LOOPBACK_BULK_EP_MPS),
```

(continues on next page)

(continued from previous page)

```
                .bInterval = 0x00,
        },

        /* Data Endpoint IN */
        .if0_in_ep = {
                .bLength = sizeof(struct usb_ep_descriptor),
                .bDescriptorType = USB_DESC_ENDPOINT,
                .bEndpointAddress = LOOPBACK_IN_EP_ADDR,
                .bmAttributes = USB_DC_EP_BULK,
                .wMaxPacketSize = sys_cpu_to_le16(CONFIG_LOOPBACK_BULK_EP_MPS),
                .bInterval = 0x00,
        },
};
```

Endpoint configuration:

```
static struct usb_ep_cfg_data ep_cfg[] = {
        {
                .ep_cb = loopback_out_cb,
                .ep_addr = LOOPBACK_OUT_EP_ADDR,
        },
        {
                .ep_cb = loopback_in_cb,
                .ep_addr = LOOPBACK_IN_EP_ADDR,
        },
};
```

USB Device configuration structure:

```
USBD_CFG_DATA_DEFINE(primary, loopback) struct usb_cfg_data loopback_config = {
        .usb_device_description = NULL,
        .interface_config = loopback_interface_config,
        .interface_descriptor = &loopback_cfg.if0,
        .cb_usb_status = loopback_status_cb,
        .interface = {
                .class_handler = NULL,
                .custom_handler = NULL,
                .vendor_handler = loopback_vendor_handler,
        },
        .num_endpoints = ARRAY_SIZE(ep_cfg),
        .endpoint = ep_cfg,
};
```

The vendor device requests are forwarded by the USB stack core driver to the class driver through the registered vendor handler.

For the loopback class driver, `loopback_vendor_handler()` processes the vendor requests:

```
static int loopback_vendor_handler(struct usb_setup_packet *setup,
                                   int32_t *len, uint8_t **data)
{
        LOG_DBG("Class request: bRequest 0x%x bmRequestType 0x%x len %d",
                setup->bRequest, setup->bmRequestType, *len);

        if (setup->RequestType.recipient != USB_REQTYPE_RECIPIENT_DEVICE) {
                return -ENOTSUP;
        }
```

(continues on next page)

(continued from previous page)

```
10
11          if (usb_reqtype_is_to_device(setup) &&
12              setup->bRequest == 0x5b) {
13                  LOG_DBG("Host-to-Device, data %p", *data);
14                  /*
15                   * Copy request data in loopback_buf buffer and reuse
16                   * it later in control device-to-host transfer.
17                   */
18                  memcpy(loopback_buf, *data,
19                         MIN(sizeof(loopback_buf), setup->wLength));
20                  return 0;
21          }
22
23          if ((usb_reqtype_is_to_host(setup)) &&
24              (setup->bRequest == 0x5c)) {
25                  LOG_DBG("Device-to-Host, wLength %d, data %p",
26                          setup->wLength, *data);
27                  *data = loopback_buf;
28                  *len = MIN(sizeof(loopback_buf), setup->wLength);
29                  return 0;
30          }
31
32          return -ENOTSUP;
33  }
```

The class driver waits for the `USB_DC_CONFIGURED` device status code before transmitting any data.

USB Vendor and Product identifiers

The USB Vendor ID for the Zephyr project is `0x2FE3`. This USB Vendor ID must not be used when a vendor integrates Zephyr USB device support into its own product.

Each USB sample has its own unique Product ID. The USB maintainer, if one is assigned, or otherwise the Zephyr Technical Steering Committee, may allocate other USB Product IDs based on well-motivated and documented requests.

When adding a new sample, add a new entry in `samples/subsys/usb/usb_pid.Kconfig` and a Kconfig file inside your sample subdirectory. The following Product IDs are currently used:

- `CONFIG_USB_PID_CDC_ACM_SAMPLE`
- `CONFIG_USB_PID_CDC_ACM_COMPOSITE_SAMPLE`
- `CONFIG_USB_PID_HID_CDC_SAMPLE`
- `CONFIG_USB_PID_CONSOLE_SAMPLE`
- `CONFIG_USB_PID_DFU_SAMPLE`
- `CONFIG_USB_PID_HID_SAMPLE`
- `CONFIG_USB_PID_HID_MOUSE_SAMPLE`
- `CONFIG_USB_PID_MASS_SAMPLE`
- `CONFIG_USB_PID_TESTUSB_SAMPLE`
- `CONFIG_USB_PID_WEBUSB_SAMPLE`
- `CONFIG_USB_PID_BLE_HCI_H4_SAMPLE`

The USB device descriptor field `bcdDevice` (Device Release Number) represents the Zephyr kernel major and minor versions as a binary coded decimal value.

API reference

There are two ways to transmit data, using the 'low' level read/write API or the 'high' level transfer API.

Low level API To transmit data to the host, the class driver should call usb_write(). Upon completion the registered endpoint callback will be called. Before sending another packet the class driver should wait for the completion of the previous write. When data is received, the registered endpoint callback is called. usb_read() should be used for retrieving the received data. For CDC ACM sample driver this happens via the OUT bulk endpoint handler (cdc_acm_bulk_out) mentioned in the endpoint array (cdc_acm_ep_data).

High level API The usb_transfer method can be used to transfer data to/from the host. The transfer API will automatically split the data transmission into one or more USB transaction(s), depending endpoint max packet size. The class driver does not have to implement endpoint callback and should set this callback to the generic usb_transfer_ep_callback.

group `_usb_device_core_api`

USB Device Core Layer API.

Defines

`USB_TRANS_READ`

`USB_TRANS_WRITE`

`USB_TRANS_NO_ZLP`

Typedefs

typedef void (*`usb_ep_callback`)(uint8_t ep, enum *usb_dc_ep_cb_status_code* cb_status)

Callback function signature for the USB Endpoint status.

typedef int (*`usb_request_handler`)(struct usb_setup_packet *setup, int32_t *transfer_len, uint8_t **payload_data)

Callback function signature for class specific requests.

Function which handles Class specific requests corresponding to an interface number specified in the device descriptor table. For host to device direction the 'len' and 'payload_data' contain the length of the received data and the pointer to the received data respectively. For device to host class requests, 'len' and 'payload_data' should be set by the callback function with the length and the address of the data to be transmitted buffer respectively.

typedef void (*`usb_interface_config`)(struct usb_desc_header *head, uint8_t bInterfaceNumber)

Function for interface runtime configuration.

typedef void (*`usb_transfer_callback`)(uint8_t ep, int tsize, void *priv)

Callback function signature for transfer completion.

Functions

int usb_set_config(const uint8_t *usb_descriptor)

Configure USB controller.

Function to configure USB controller. Configuration parameters must be valid or an error is returned

Parameters

- usb_descriptor – **[in]** USB descriptor table

Returns 0 on success, negative errno code on fail

int usb_deconfig(void)

Deconfigure USB controller.

This function returns the USB device to it's initial state

Returns 0 on success, negative errno code on fail

int usb_enable(*usb_dc_status_callback* status_cb)

Enable the USB subsystem and associated hardware.

This function initializes the USB core subsystem and enables the corresponding hardware so that it can begin transmitting and receiving on the USB bus, as well as generating interrupts.

Class-specific initialization and registration must be performed by the user before invoking this, so that any data or events on the bus are processed correctly by the associated class handling code.

Parameters

- status_cb – **[in]** Callback registered by user to notify about USB device controller state.

Returns 0 on success, negative errno code on fail.

int usb_disable(void)

Disable the USB device.

Function to disable the USB device. Upon success, the specified USB interface is clock gated in hardware, it is no longer capable of generating interrupts.

Returns 0 on success, negative errno code on fail

int usb_write(uint8_t ep, const uint8_t *data, uint32_t data_len, uint32_t *bytes_ret)

Write data to the specified endpoint.

Function to write data to the specified endpoint. The supplied usb_ep_callback will be called when transmission is done.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- data – **[in]** Pointer to data to write
- data_len – **[in]** Length of data requested to write. This may be zero for a zero length status packet.
- bytes_ret – **[out]** Bytes written to the EP FIFO. This value may be NULL if the application expects all bytes to be written

Returns 0 on success, negative errno code on fail

int `usb_read`(uint8_t ep, uint8_t *data, uint32_t max_data_len, uint32_t *ret_bytes)

Read data from the specified endpoint.

This function is called by the Endpoint handler function, after an OUT interrupt has been received for that EP. The application must only call this function through the supplied usb_ep_callback function.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `data` – **[in]** Pointer to data buffer to write to
- `max_data_len` – **[in]** Max length of data to read
- `ret_bytes` – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read is returned.

Returns 0 on success, negative errno code on fail

int `usb_ep_set_stall`(uint8_t ep)

Set STALL condition on the specified endpoint.

This function is called by USB device class handler code to set stall condition on endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail

int `usb_ep_clear_stall`(uint8_t ep)

Clears STALL condition on the specified endpoint.

This function is called by USB device class handler code to clear stall condition on endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail

int `usb_ep_read_wait`(uint8_t ep, uint8_t *data, uint32_t max_data_len, uint32_t *read_bytes)

Read data from the specified endpoint.

This is similar to usb_ep_read, the difference being that, it doesn't clear the endpoint NAKs so that the consumer is not bogged down by further upcalls till he is done with the processing of the data. The caller should reactivate ep by invoking *usb_ep_read_continue()* do so.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- `data` – **[in]** pointer to data buffer to write to
- `max_data_len` – **[in]** max length of data to read
- `read_bytes` – **[out]** Number of bytes read. If data is NULL and max_data_len is 0 the number of bytes available for read should be returned.

Returns 0 on success, negative errno code on fail.

int usb_ep_read_continue(uint8_t ep)

Continue reading data from the endpoint.

Clear the endpoint NAK and enable the endpoint to accept more data from the host. Usually called after *usb_ep_read_wait()* when the consumer is fine to accept more data. Thus these calls together acts as flow control mechanism.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

void usb_transfer_ep_callback(uint8_t ep, enum *usb_dc_ep_cb_status_code*)

Transfer management endpoint callback.

If a USB class driver wants to use high-level transfer functions, driver needs to register this callback as usb endpoint callback.

int usb_transfer(uint8_t ep, uint8_t *data, size_t dlen, unsigned int flags, *usb_transfer_callback* cb, void *priv)

Start a transfer.

Start a usb transfer to/from the data buffer. This function is asynchronous and can be executed in IRQ context. The provided callback will be called on transfer completion (or error) in thread context.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- data – **[in]** Pointer to data buffer to write-to/read-from
- dlen – **[in]** Size of data buffer
- flags – **[in]** Transfer flags (USB_TRANS_READ, USB_TRANS_WRITE...)
- cb – **[in]** Function called on transfer completion/failure
- priv – **[in]** Data passed back to the transfer completion callback

Returns 0 on success, negative errno code on fail.

int usb_transfer_sync(uint8_t ep, uint8_t *data, size_t dlen, unsigned int flags)

Start a transfer and block-wait for completion.

Synchronous version of usb_transfer, wait for transfer completion before returning.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table
- data – **[in]** Pointer to data buffer to write-to/read-from
- dlen – **[in]** Size of data buffer
- flags – **[in]** Transfer flags

Returns number of bytes transferred on success, negative errno code on fail.

void usb_cancel_transfer(uint8_t ep)

Cancel any ongoing transfer on the specified endpoint.

Parameters

- ep – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns 0 on success, negative errno code on fail.

void `usb_cancel_transfers`(void)

Cancel all ongoing transfers.

bool `usb_transfer_is_busy`(uint8_t ep)

Check that transfer is ongoing for the endpoint.

Parameters

- `ep` – **[in]** Endpoint address corresponding to the one listed in the device configuration table

Returns true if transfer is ongoing, false otherwise.

int `usb_wakeup_request`(void)

Start the USB remote wakeup procedure.

Function to request a remote wakeup. This feature must be enabled in configuration, otherwise it will always return -ENOTSUP error.

Returns 0 on success, negative errno code on fail, i.e. when the bus is already active.

struct `usb_ep_cfg_data`

#include <usb_device.h> USB Endpoint Configuration.

This structure contains configuration for the endpoint.

Public Members

usb_ep_callback `ep_cb`

Callback function for notification of data received and available to application or transmit done, NULL if callback not required by application code

uint8_t `ep_addr`

The number associated with the EP in the device configuration structure IN EP = 0x80 | <endpoint number> OUT EP = 0x00 | <endpoint number>

struct `usb_interface_cfg_data`

#include <usb_device.h> USB Interface Configuration.

This structure contains USB interface configuration.

Public Members

usb_request_handler `class_handler`

Handler for USB Class specific Control (EP 0) communications

usb_request_handler `vendor_handler`

Handler for USB Vendor specific commands

usb_request_handler `custom_handler`

The custom request handler gets a first chance at handling the request before it is handed over to the 'chapter 9' request handler. return 0 on success, -EINVAL if the request has not been handled by the custom handler and instead needs to be handled by the core USB stack. Any other error code to denote failure within the custom handler.

struct usb_cfg_data

#include <usb_device.h> USB device configuration.

The Application instantiates this with given parameters added using the "usb_set_config" function. Once this function is called changes to this structure will result in undefined behavior. This structure may only be updated after calls to usb_deconfig

Public Members

const uint8_t *usb_device_description

USB device description, see http://www.beyondlogic.org/usbnutshell/usb5.shtml#DeviceDescriptors

void *interface_descriptor

Pointer to interface descriptor

usb_interface_config interface_config

Function for interface runtime configuration

void (*cb_usb_status)(struct *usb_cfg_data* *cfg, enum *usb_dc_status_code* cb_status, const uint8_t *param)

Callback to be notified on USB connection status change

struct *usb_interface_cfg_data* interface

USB interface (Class) handler and storage space

uint8_t num_endpoints

Number of individual endpoints in the device configuration

struct *usb_ep_cfg_data* *endpoint

Pointer to an array of endpoint structs of length equal to the number of EP associated with the device description, not including control endpoints

7.29.3 Testing USB device support

Testing over USPIP in native_posix

A virtual USB controller implemented through USBIP might be used to test the USB Device stack. Follow the general build procedure to build the USB sample for the native_posix configuration.

Run built sample with:

```
west build -t run
```

In a terminal window, run the following command to list USB devices:

```
$ usbip list -r localhost
Exportable USB devices
======================
 - 127.0.0.1
        1-1: unknown vendor : unknown product (2fe3:0100)
           : /sys/devices/pci0000:00/0000:00:01.2/usb1/1-1
```

(continues on next page)

(continued from previous page)

```
           : (Defined at Interface level) (00/00/00)
           :  0 - Vendor Specific Class / unknown subclass / unknown protocol (ff/00/
↪00)
```

In a terminal window, run the following command to attach the USB device:

```
$ sudo usbip attach -r localhost -b 1-1
```

The USB device should be connected to your Linux host, and verified with the following commands:

```
$ sudo usbip port
Imported USB devices
====================
Port 00: <Port in Use> at Full Speed(12Mbps)
       unknown vendor : unknown product (2fe3:0100)
       7-1 -> usbip://localhost:3240/1-1
           -> remote bus/dev 001/002
$ lsusb -d 2fe3:0100
Bus 007 Device 004: ID 2fe3:0100
```

7.29.4 USB Human Interface Devices (HID) support

Since the USB HID specification is not only used by the USB subsystem, the USB HID API is split into two header files include/usb/class/hid.h and include/usb/class/usb_hid.h. The second includes a specific part for HID support in the USB device stack.

HID Item helpers

HID item helper macros can be used to compose a HID Report Descriptor. The names correspond to those used in the USB HID Specification.

Example of a HID Report Descriptor:

```
static const uint8_t hid_report_desc[] = {
    HID_USAGE_PAGE(HID_USAGE_GEN_DESKTOP),
    HID_USAGE(HID_USAGE_GEN_DESKTOP_UNDEFINED),
    HID_COLLECTION(HID_COLLECTION_APPLICATION),
    HID_LOGICAL_MIN8(0),
    /* logical maximum 255 */
    HID_LOGICAL_MAX16(0xFF, 0x00),
    HID_REPORT_ID(1),
    HID_REPORT_SIZE(8),
    HID_REPORT_COUNT(1),
    HID_USAGE(HID_USAGE_GEN_DESKTOP_UNDEFINED),
    /* HID_INPUT (Data, Variable, Absolute) */
    HID_INPUT(0x02),
    HID_END_COLLECTION,
};
```

HID items reference

group usb_hid_items

Defines

`HID_ITEM(bTag, bType, bSize)`

Define HID short item.

Parameters

- `bTag` – Item tag
- `bType` – Item type
- `bSize` – Item data size

Returns HID Input item

`HID_INPUT(a)`

Define HID Input item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Input item data

Returns HID Input item

`HID_OUTPUT(a)`

Define HID Output item with the data length of one byte.

For usage examples, see *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Output item data

Returns HID Output item

`HID_FEATURE(a)`

Define HID Feature item with the data length of one byte.

Parameters

- `a` – Feature item data

Returns HID Feature item

`HID_COLLECTION(a)`

Define HID Collection item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Collection item data

Returns HID Collection item

`HID_END_COLLECTION`

Define HID End Collection (non-data) item.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Returns HID End Collection item

`HID_USAGE_PAGE(page)`

Define HID Usage Page item.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- page – Usage Page

Returns HID Usage Page item

`HID_LOGICAL_MIN8(a)`

Define HID Logical Minimum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- a – Minimum value in logical units

Returns HID Logical Minimum item

`HID_LOGICAL_MAX8(a)`

Define HID Logical Maximum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- a – Maximum value in logical units

Returns HID Logical Maximum item

`HID_LOGICAL_MIN16(a, b)`

Define HID Logical Minimum item with the data length of two bytes.

Parameters

- a – Minimum value lower byte
- b – Minimum value higher byte

Returns HID Logical Minimum item

`HID_LOGICAL_MAX16(a, b)`

Define HID Logical Maximum item with the data length of two bytes.

Parameters

- a – Minimum value lower byte
- b – Minimum value higher byte

Returns HID Logical Maximum item

`HID_LOGICAL_MIN32(a, b, c, d)`

Define HID Logical Minimum item with the data length of four bytes.

Parameters

- a – Minimum value lower byte
- b – Minimum value low middle byte
- c – Minimum value high middle byte
- d – Minimum value higher byte

Returns HID Logical Minimum item

`HID_LOGICAL_MAX32(a, b, c, d)`

Define HID Logical Maximum item with the data length of four bytes.

Parameters

- a – Minimum value lower byte
- b – Minimum value low middle byte
- c – Minimum value high middle byte

- `d` – Minimum value higher byte

Returns HID Logical Maximum item

`HID_REPORT_SIZE(size)`

Define HID Report Size item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `size` – Report field size in bits

Returns HID Report Size item

`HID_REPORT_ID(id)`

Define HID Report ID item with the data length of one byte.

Parameters

- `id` – Report ID

Returns HID Report ID item

`HID_REPORT_COUNT(count)`

Define HID Report Count item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `count` – Number of data fields included in the report

Returns HID Report Count item

`HID_USAGE(idx)`

Define HID Usage Index item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `idx` – Number of data fields included in the report

Returns HID Usage Index item

`HID_USAGE_MIN8(a)`

Define HID Usage Minimum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Starting Usage

Returns HID Usage Minimum item

`HID_USAGE_MAX8(a)`

Define HID Usage Maximum item with the data length of one byte.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Ending Usage

Returns HID Usage Maximum item

`HID_USAGE_MIN16(a, b)`

Define HID Usage Minimum item with the data length of two bytes.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Starting Usage lower byte
- `b` – Starting Usage higher byte

Returns HID Usage Minimum item

`HID_USAGE_MAX16(a, b)`

Define HID Usage Maximum item with the data length of two bytes.

For usage examples, see *HID_MOUSE_REPORT_DESC()*, *HID_KEYBOARD_REPORT_DESC()*

Parameters

- `a` – Ending Usage lower byte
- `b` – Ending Usage higher byte

Returns HID Usage Maximum item

HID types reference

group `usb_hid_types`

Defines

`USB_DESC_HID`

USB HID Class HID descriptor type

`USB_DESC_HID_REPORT`

USB HID Class Report descriptor type

`USB_DESC_HID_PHYSICAL`

USB HID Class physical descriptor type

`USB_HID_GET_REPORT`

USB HID Class GetReport bRequest value

`USB_HID_GET_IDLE`

USB HID Class GetIdle bRequest value

`USB_HID_GET_PROTOCOL`

USB HID Class GetProtocol bRequest value

`USB_HID_SET_REPORT`

USB HID Class SetReport bRequest value

`USB_HID_SET_IDLE`

USB HID Class SetIdle bRequest value

USB_HID_SET_PROTOCOL
: USB HID Class SetProtocol bRequest value

HID_BOOT_IFACE_CODE_NONE
: USB HID Boot Interface Protocol (bInterfaceProtocol) Code None

HID_BOOT_IFACE_CODE_KEYBOARD
: USB HID Boot Interface Protocol (bInterfaceProtocol) Code Keyboard

HID_BOOT_IFACE_CODE_MOUSE
: USB HID Boot Interface Protocol (bInterfaceProtocol) Code Mouse

HID_PROTOCOL_BOOT
: USB HID Class Boot protocol code

HID_PROTOCOL_REPORT
: USB HID Class Report protocol code

HID_ITEM_TYPE_MAIN
: HID Main item type

HID_ITEM_TYPE_GLOBAL
: HID Global item type

HID_ITEM_TYPE_LOCAL
: HID Local item type

HID_ITEM_TAG_INPUT
: HID Input item tag

HID_ITEM_TAG_OUTPUT
: HID Output item tag

HID_ITEM_TAG_COLLECTION
: HID Collection item tag

HID_ITEM_TAG_FEATURE
: HID Feature item tag

HID_ITEM_TAG_COLLECTION_END
: HID End Collection item tag

HID_ITEM_TAG_USAGE_PAGE
: HID Usage Page item tag

HID_ITEM_TAG_LOGICAL_MIN
: HID Logical Minimum item tag

HID_ITEM_TAG_LOGICAL_MAX
: HID Logical Maximum item tag

HID_ITEM_TAG_PHYSICAL_MIN
: HID Physical Minimum item tag

HID_ITEM_TAG_PHYSICAL_MAX
: HID Physical Maximum item tag

HID_ITEM_TAG_UNIT_EXPONENT
: HID Unit Exponent item tag

HID_ITEM_TAG_UNIT
: HID Unit item tag

HID_ITEM_TAG_REPORT_SIZE
: HID Report Size item tag

HID_ITEM_TAG_REPORT_ID
: HID Report ID item tag

HID_ITEM_TAG_REPORT_COUNT
: HID Report count item tag

HID_ITEM_TAG_USAGE
: HID Usage item tag

HID_ITEM_TAG_USAGE_MIN
: HID Usage Minimum item tag

HID_ITEM_TAG_USAGE_MAX
: HID Usage Maximum item tag

HID_COLLECTION_PHYSICAL
: Physical collection type

HID_COLLECTION_APPLICATION
: Application collection type

HID_USAGE_GEN_DESKTOP
: HID Generic Desktop Controls Usage page

HID_USAGE_GEN_KEYBOARD
: HID Keyboard Usage page

HID_USAGE_GEN_LEDS
: HID LEDs Usage page

HID_USAGE_GEN_BUTTON
: HID Button Usage page

HID_USAGE_GEN_DESKTOP_UNDEFINED
: HID Generic Desktop Undefined Usage ID

HID_USAGE_GEN_DESKTOP_POINTER
: HID Generic Desktop Pointer Usage ID

HID_USAGE_GEN_DESKTOP_MOUSE
: HID Generic Desktop Mouse Usage ID

HID_USAGE_GEN_DESKTOP_JOYSTICK
: HID Generic Desktop Joystick Usage ID

HID_USAGE_GEN_DESKTOP_GAMEPAD
: HID Generic Desktop Gamepad Usage ID

HID_USAGE_GEN_DESKTOP_KEYBOARD
: HID Generic Desktop Keyboard Usage ID

HID_USAGE_GEN_DESKTOP_KEYPAD
: HID Generic Desktop Keypad Usage ID

HID_USAGE_GEN_DESKTOP_X
: HID Generic Desktop X Usage ID

HID_USAGE_GEN_DESKTOP_Y
: HID Generic Desktop Y Usage ID

HID_USAGE_GEN_DESKTOP_WHEEL
: HID Generic Desktop Wheel Usage ID

HID Mouse and Keyboard report descriptors

The pre-defined Mouse and Keyboard report descriptors can be used by a HID device implementation or simply as examples.

group usb_hid_mk_report_desc

Defines

HID_MOUSE_REPORT_DESC(bcnt)
: Simple HID mouse report descriptor for n button mouse.

  Parameters

  - bcnt – Button count. Allowed values from 1 to 8.

HID_KEYBOARD_REPORT_DESC()
: Simple HID keyboard report descriptor.

Enums

enum `hid_kbd_code`

HID keyboard button codes.

Values:

enumerator `HID_KEY_A` = 4

enumerator `HID_KEY_B` = 5

enumerator `HID_KEY_C` = 6

enumerator `HID_KEY_D` = 7

enumerator `HID_KEY_E` = 8

enumerator `HID_KEY_F` = 9

enumerator `HID_KEY_G` = 10

enumerator `HID_KEY_H` = 11

enumerator `HID_KEY_I` = 12

enumerator `HID_KEY_J` = 13

enumerator `HID_KEY_K` = 14

enumerator `HID_KEY_L` = 15

enumerator `HID_KEY_M` = 16

enumerator `HID_KEY_N` = 17

enumerator `HID_KEY_O` = 18

enumerator `HID_KEY_P` = 19

enumerator `HID_KEY_Q` = 20

enumerator `HID_KEY_R` = 21

enumerator `HID_KEY_S` = 22

enumerator `HID_KEY_T` = 23

enumerator HID_KEY_U = 24

enumerator HID_KEY_V = 25

enumerator HID_KEY_W = 26

enumerator HID_KEY_X = 27

enumerator HID_KEY_Y = 28

enumerator HID_KEY_Z = 29

enumerator HID_KEY_1 = 30

enumerator HID_KEY_2 = 31

enumerator HID_KEY_3 = 32

enumerator HID_KEY_4 = 33

enumerator HID_KEY_5 = 34

enumerator HID_KEY_6 = 35

enumerator HID_KEY_7 = 36

enumerator HID_KEY_8 = 37

enumerator HID_KEY_9 = 38

enumerator HID_KEY_0 = 39

enumerator HID_KEY_ENTER = 40

enumerator HID_KEY_ESC = 41

enumerator HID_KEY_BACKSPACE = 42

enumerator HID_KEY_TAB = 43

enumerator HID_KEY_SPACE = 44

enumerator HID_KEY_MINUS = 45

enumerator HID_KEY_EQUAL = 46

enumerator HID_KEY_LEFTBRACE = 47

enumerator HID_KEY_RIGHTBRACE = 48

enumerator HID_KEY_BACKSLASH = 49

enumerator HID_KEY_HASH = 50

enumerator HID_KEY_SEMICOLON = 51

enumerator HID_KEY_APOSTROPHE = 52

enumerator HID_KEY_GRAVE = 53

enumerator HID_KEY_COMMA = 54

enumerator HID_KEY_DOT = 55

enumerator HID_KEY_SLASH = 56

enumerator HID_KEY_CAPSLOCK = 57

enumerator HID_KEY_F1 = 58

enumerator HID_KEY_F2 = 59

enumerator HID_KEY_F3 = 60

enumerator HID_KEY_F4 = 61

enumerator HID_KEY_F5 = 62

enumerator HID_KEY_F6 = 63

enumerator HID_KEY_F7 = 64

enumerator HID_KEY_F8 = 65

enumerator HID_KEY_F9 = 66

enumerator HID_KEY_F10 = 67

enumerator HID_KEY_F11 = 68

enumerator HID_KEY_F12 = 69

enumerator HID_KEY_SYSRQ = 70

enumerator HID_KEY_SCROLLLOCK = 71

enumerator HID_KEY_PAUSE = 72

enumerator HID_KEY_INSERT = 73

enumerator HID_KEY_HOME = 74

enumerator HID_KEY_PAGEUP = 75

enumerator HID_KEY_DELETE = 76

enumerator HID_KEY_END = 77

enumerator HID_KEY_PAGEDOWN = 78

enumerator HID_KEY_RIGHT = 79

enumerator HID_KEY_LEFT = 80

enumerator HID_KEY_DOWN = 81

enumerator HID_KEY_UP = 82

enumerator HID_KEY_NUMLOCK = 83

enumerator HID_KEY_KPSLASH = 84

enumerator HID_KEY_KPASTERISK = 85

enumerator HID_KEY_KPMINUS = 86

enumerator HID_KEY_KPPLUS = 87

enumerator HID_KEY_KPENTER = 88

enumerator HID_KEY_KP_1 = 89

enumerator HID_KEY_KP_2 = 90

enumerator HID_KEY_KP_3 = 91

enumerator HID_KEY_KP_4 = 92

enumerator HID_KEY_KP_5 = 93

enumerator HID_KEY_KP_6 = 94

enumerator HID_KEY_KP_7 = 95

enumerator HID_KEY_KP_8 = 96

enumerator HID_KEY_KP_9 = 97

enumerator HID_KEY_KP_0 = 98

enum hid_kbd_modifier

HID keyboard modifiers.

Values:

enumerator HID_KBD_MODIFIER_NONE = 0x00

enumerator HID_KBD_MODIFIER_LEFT_CTRL = 0x01

enumerator HID_KBD_MODIFIER_LEFT_SHIFT = 0x02

enumerator HID_KBD_MODIFIER_LEFT_ALT = 0x04

enumerator HID_KBD_MODIFIER_LEFT_UI = 0x08

enumerator HID_KBD_MODIFIER_RIGHT_CTRL = 0x10

enumerator HID_KBD_MODIFIER_RIGHT_SHIFT = 0x20

enumerator HID_KBD_MODIFIER_RIGHT_ALT = 0x40

enumerator HID_KBD_MODIFIER_RIGHT_UI = 0x80

enum hid_kbd_led

HID keyboard LEDs.

Values:

enumerator HID_KBD_LED_NUM_LOCK = 0x01

enumerator HID_KBD_LED_CAPS_LOCK = 0x02

enumerator HID_KBD_LED_SCROLL_LOCK = 0x04

enumerator HID_KBD_LED_COMPOSE = 0x08

enumerator HID_KBD_LED_KANA = 0x10

HID Class Device API reference

USB HID devices like mouse, keyboard, or any other specific device use this API.

group usb_hid_device_api

Typedefs

typedef int (*hid_cb_t)(const struct *device* *dev, struct usb_setup_packet *setup, int32_t *len, uint8_t **data)

typedef void (*hid_int_ready_callback)(const struct *device* *dev)

typedef void (*hid_protocol_cb_t)(const struct *device* *dev, uint8_t protocol)

typedef void (*hid_idle_cb_t)(const struct *device* *dev, uint16_t report_id)

Functions

void usb_hid_register_device(const struct *device* *dev, const uint8_t *desc, size_t size, const struct *hid_ops* *op)

Register HID device.

Parameters

- dev – **[in]** Pointer to USB HID device
- desc – **[in]** Pointer to HID report descriptor
- size – **[in]** Size of HID report descriptor
- op – **[in]** Pointer to USB HID device interrupt struct

int hid_int_ep_write(const struct *device* *dev, const uint8_t *data, uint32_t data_len, uint32_t *bytes_ret)

Write to USB HID interrupt endpoint buffer.

Parameters

- dev – **[in]** Pointer to USB HID device
- data – **[in]** Pointer to data buffer
- data_len – **[in]** Length of data to copy
- bytes_ret – **[out]** Bytes written to the EP buffer.

Returns 0 on success, negative errno code on fail.

int hid_int_ep_read(const struct *device* *dev, uint8_t *data, uint32_t max_data_len, uint32_t *ret_bytes)

Read from USB HID interrupt endpoint buffer.

Parameters

- dev – **[in]** Pointer to USB HID device

- `data` – **[in]** Pointer to data buffer
- `max_data_len` – **[in]** Max length of data to copy
- `ret_bytes` – **[out]** Number of bytes to copy. If data is NULL and ret_bytes is 0 the number of bytes available in the buffer will be returned.

Returns 0 on success, negative errno code on fail.

int `usb_hid_set_proto_code`(const struct *device* *dev, uint8_t proto_code)

Set USB HID class Protocol Code.

Should be called before *usb_hid_init()*.

Parameters

- `dev` – **[in]** Pointer to USB HID device
- `proto_code` – **[in]** Protocol Code to be used for bInterfaceProtocol

Returns 0 on success, negative errno code on fail.

int `usb_hid_init`(const struct *device* *dev)

Initialize USB HID class support.

Parameters

- `dev` – **[in]** Pointer to USB HID device

Returns 0 on success, negative errno code on fail.

struct `hid_ops`

#include <usb_hid.h> USB HID device interface.

7.29.5 USB device stack CDC ACM support

The CDC ACM class is used as backend for different subsystems in Zephyr. However, its configuration may not be easy for the inexperienced user. Below is a description of the different use cases and some pitfalls.

The interface for CDC ACM user is *UART* driver API. But there are two important differences in behavior to a real UART controller:

- Data transfer is only possible after the USB device stack has been initialized and started, until then any data is discarded
- If device is connected to the host, it still needs an application on the host side which requests the data

The devicetree compatible property for CDC ACM UART is `zephyr,cdc-acm-uart`. CDC ACM support is automatically selected when USB device support is enabled and a compatible node in the devicetree sources is present. If necessary, CDC ACM support can be explicitly disabled by `CONFIG_USB_CDC_ACM`. About four CDC ACM UART instances can be defined and used, limited by the maximum number of supported endpoints on the controller.

CDC ACM UART node is supposed to be child of a USB device controller node. Since the designation of the controller nodes varies from vendor to vendor, and our samples and application should be as generic as possible, the default USB device controller is usually assigned an `zephyr_udc0` node label. Often, CDC ACM UART is described in a devicetree overlay file and looks like this:

```
&zephyr_udc0 {
     cdc_acm_uart0: cdc_acm_uart0 {
             compatible = "zephyr,cdc-acm-uart";
             label = "CDC_ACM_0";
```

(continues on next page)

(continued from previous page)

```
        };
};
```

Samples usb_cdc-acm and usb_hid-cdc have similar overlay files. And since no special properties are present, it may seem overkill to use devicetree to describe CDC ACM UART. The motivation behind using devicetree is the easy interchangeability of a real UART controller and CDC ACM UART in applications.

Console over CDC ACM UART

With the CDC ACM UART node from above and `zephyr,console` property of the chosen node, we can describe that CDC ACM UART is to be used with the console. A similar overlay file is used by cdc-acm-console. If USB device support is enabled in the application, as in the console sample, `CONFIG_USB_UART_CONSOLE` must be enabled, which does nothing but change the initialization time of the console driver.

```
/ {
        chosen {
                zephyr,console = &cdc_acm_uart0;
        };
};

&zephyr_udc0 {
        cdc_acm_uart0: cdc_acm_uart0 {
                compatible = "zephyr,cdc-acm-uart";
                label = "CDC_ACM_0";
        };
};
```

Before the application uses the console, it is recommended to wait for the DTR signal:

```
const struct device *dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_console));
uint32_t dtr = 0;

if (usb_enable(NULL)) {
    return;
}

while (!dtr) {
    uart_line_ctrl_get(dev, UART_LINE_CTRL_DTR, &dtr);
    k_sleep(K_MSEC(100));
}

printk("nuqneH\n");
```

CDC ACM UART as backend

As for the console sample, it is possible to configure CDC ACM UART as backend for other subsystems by setting *Chosen nodes* properties.

List of few Zephyr specific chosen properties which can be used to select CDC ACM UART as backend for a subsystem or application:

- `zephyr,bt-c2h-uart` used in Bluetooth, for example see bluetooth-hci-uart-sample
- `zephyr,ot-uart` used in OpenThread, for example see coprocessor-sample

- `zephyr,shell-uart` used by shell for serial backend, for example see samples/subsys/shell/shell_module
- `zephyr,uart-mcumgr` used by smp_svr_sample

7.30 User Mode

Zephyr offers the capability to run threads at a reduced privilege level which we call user mode. The current implementation is designed for devices with MPU hardware.

For details on creating threads that run in user mode, please see *Lifecycle*.

7.30.1 Overview

Threat Model

User mode threads are considered to be untrusted by Zephyr and are therefore isolated from other user mode threads and from the kernel. A flawed or malicious user mode thread cannot leak or modify the private data/resources of another thread or the kernel, and cannot interfere with or control another user mode thread or the kernel.

Example use-cases of Zephyr's user mode features:

- The kernel can protect against many unintentional programming errors which could otherwise silently or spectacularly corrupt the system.
- The kernel can sandbox complex data parsers such as interpreters, network protocols, and filesystems such that malicious third-party code or data cannot compromise the kernel or other threads.
- The kernel can support the notion of multiple logical "applications", each with their own group of threads and private data structures, which are isolated from each other if one crashes or is otherwise compromised.

Design Goals For threads running in a non-privileged CPU state (hereafter referred to as 'user mode') we aim to protect against the following:

- We prevent access to memory not specifically granted, or incorrect access to memory that has an incompatible policy, such as attempting to write to a read-only area.
  - Access to thread stack buffers will be controlled with a policy which partially depends on the underlying memory protection hardware.
    - A user thread will by default have read/write access to its own stack buffer.
    - A user thread will never by default have access to user thread stacks that are not members of the same memory domain.
    - A user thread will never by default have access to thread stacks owned by a supervisor thread, or thread stacks used to handle system call privilege elevations, interrupts, or CPU exceptions.
    - A user thread may have read/write access to the stacks of other user threads in the same memory domain, depending on hardware.
      - On MPU systems, threads may only access their own stack buffer.
      - On MMU systems, threads may access any user thread stack in the same memory domain. Portable code should not assume this.
  - By default, program text and read-only data are accessible to all threads on read-only basis, kernel-wide. This policy may be adjusted.

  - User threads by default are not granted default access to any memory except what is noted above.

- We prevent use of device drivers or kernel objects not specifically granted, with the permission granularity on a per object or per driver instance basis.
- We validate kernel or driver API calls with incorrect parameters that would otherwise cause a crash or corruption of data structures private to the kernel. This includes:
  - Using the wrong kernel object type.
  - Using parameters outside of proper bounds or with nonsensical values.
  - Passing memory buffers that the calling thread does not have sufficient access to read or write, depending on the semantics of the API.
  - Use of kernel objects that are not in a proper initialization state.
- We ensure the detection and safe handling of user mode stack overflows.
- We prevent invoking system calls to functions excluded by the kernel configuration.
- We prevent disabling of or tampering with kernel-defined and hardware- enforced memory protections.
- We prevent re-entry from user to supervisor mode except through the kernel- defined system calls and interrupt handlers.
- We prevent the introduction of new executable code by user mode threads, except to the extent to which this is supported by kernel system calls.

We are specifically not protecting against the following attacks:

- The kernel itself, and any threads that are executing in supervisor mode, are assumed to be trusted.
- The toolchain and any supplemental programs used by the build system are assumed to be trusted.
- The kernel build is assumed to be trusted. There is considerable build-time logic for creating the tables of valid kernel objects, defining system calls, and configuring interrupts. The .elf binary files that are worked with during this process are all assumed to be trusted code.
- We can't protect against mistakes made in memory domain configuration done in kernel mode that exposes private kernel data structures to a user thread. RAM for kernel objects should always be configured as supervisor-only.
- It is possible to make top-level declarations of user mode threads and assign them permissions to kernel objects. In general, all C and header files that are part of the kernel build producing zephyr.elf are assumed to be trusted.
- We do not protect against denial of service attacks through thread CPU starvation. Zephyr has no thread priority aging and a user thread of a particular priority can starve all threads of lower priority, and also other threads of the same priority if time-slicing is not enabled.
- There are build-time defined limits on how many threads can be active simultaneously, after which creation of new user threads will fail.
- Stack overflows for threads running in supervisor mode may be caught, but the integrity of the system cannot be guaranteed.

High-level Policy Details

Broadly speaking, we accomplish these thread-level memory protection goals through the following mechanisms:

- Any user thread will only have access to a subset of memory: typically its stack, program text, read-only data, and any partitions configured in the *Memory Protection Design* it belongs to. Access to any other RAM must be done on the thread's behalf through system calls, or specifically granted by a supervisor thread using the memory domain APIs. Newly created threads inherit the memory

domain configuration of the parent. Threads may communicate with each other by having shared membership of the same memory domains, or via kernel objects such as semaphores and pipes.

- User threads cannot directly access memory belonging to kernel objects. Although pointers to kernel objects are used to reference them, actual manipulation of kernel objects is done through system call interfaces. Device drivers and threads stacks are also considered kernel objects. This ensures that any data inside a kernel object that is private to the kernel cannot be tampered with.
- User threads by default have no permission to access any kernel object or driver other than their own thread object. Such access must be granted by another thread that is either in supervisor mode or has permission on both the receiving thread object and the kernel object being granted access to. The creation of new threads has an option to automatically inherit permissions of all kernel objects granted to the parent, except the parent thread itself.
- For performance and footprint reasons Zephyr normally does little or no parameter error checking for kernel object or device driver APIs. Access from user mode through system calls involves an extra layer of handler functions, which are expected to rigorously validate access permissions and type of the object, check the validity of other parameters through bounds checking or other means, and verify proper read/write access to any memory buffers involved.
- Thread stacks are defined in such a way that exceeding the specified stack space will generate a hardware fault. The way this is done specifically varies per architecture.

Constraints

All kernel objects, thread stacks, and device driver instances must be defined at build time if they are to be used from user mode. Dynamic use-cases for kernel objects will need to go through pre-defined pools of available objects.

There are some constraints if additional application binary data is loaded for execution after the kernel starts:

- Loaded object code will not be able to define any kernel objects that will be recognized by the kernel. This code will instead need to use APIs for requesting kernel objects from pools.
- Similarly, since the loaded object code will not be part of the kernel build process, this code will not be able to install interrupt handlers, instantiate device drivers, or define system calls, regardless of what mode it runs in.
- Loaded object code that does not come from a verified source should always be entered with the CPU already in user mode.

7.30.2 Memory Protection Design

Zephyr's memory protection design is geared towards microcontrollers with MPU (Memory Protection Unit) hardware. We do support some architectures, such as x86, which have a paged MMU (Memory Management Unit), but in that case the MMU is used like an MPU with an identity page table.

All of the discussion below will be using MPU terminology; systems with MMUs can be considered to have an MPU with an unlimited number of programmable regions.

There are a few different levels on how memory access is configured when Zephyr memory protection features are enabled, which we will describe here:

Boot Time Memory Configuration

This is the configuration of the MPU after the kernel has started up. It should contain the following:

- Any configuration of memory regions which need to have special caching or write-back policies for basic hardware and driver function. Note that most MPUs have the concept of a default memory access policy map, which can be enabled as a "background" mapping for any area of memory that

doesn't have an MPU region configuring it. It is strongly recommended to use this to maximize the number of available MPU regions for the end user. On ARMv7-M/ARMv8-M this is called the System Address Map, other CPUs may have similar capabilities.

- A read-only, executable region or regions for program text and ro-data, that is accessible to user mode. This could be further sub-divided into a read-only region for ro-data, and a read-only, executable region for text, but this will require an additional MPU region. This is required so that threads running in user mode can read ro-data and fetch instructions.
- Depending on configuration, user-accessible read-write regions to support extra features like GCOV, HEP, etc.

Assuming there is a background map which allows supervisor mode to access any memory it needs, and regions are defined which grant user mode access to text/ro-data, this is sufficient for the boot time configuration.

Hardware Stack Overflow

`CONFIG_HW_STACK_PROTECTION` is an optional feature which detects stack buffer overflows when the system is running in supervisor mode. This catches issues when the entire stack buffer has overflowed, and not individual stack frames, use compiler-assisted `CONFIG_STACK_CANARIES` for that.

Like any crash in supervisor mode, no guarantees can be made about the overall health of the system after a supervisor mode stack overflow, and any instances of this should be treated as a serious error. However it's still very useful to know when these overflows happen, as without robust detection logic the system will either crash in mysterious ways or behave in an undefined manner when the stack buffer overflows.

Some systems implement this feature by creating at runtime a 'guard' MPU region which is set to be read-only and is at either the beginning or immediately preceding the supervisor mode stack buffer. If the stack overflows an exception will be generated.

This feature is optional and is not required to catch stack overflows in user mode; disabling this may free 1-2 MPU regions depending on the MPU design.

Other systems may have dedicated CPU support for catching stack overflows and no extra MPU regions will be required.

Thread Stack

Any thread running in user mode will need access to its own stack buffer. On context switch into a user mode thread, a dedicated MPU region will be programmed with the bounds of the stack buffer. A thread exceeding its stack buffer will start pushing data onto memory it doesn't have access to and a memory access violation exception will be generated.

Thread Resource Pools

A small subset of kernel APIs, invoked as system calls, require heap memory allocations. This memory is used only by the kernel and is not accessible directly by user mode. In order to use these system calls, invoking threads must assign themselves to a resource pool, which is a k_mem_pool object. Memory is drawn from a thread's resource pool using `z_thread_malloc()` and freed with *k_free()*.

The APIs which use resource pools are as follows, with any alternatives noted for users who do not want heap allocations within their application:

- *k_stack_alloc_init()* sets up a k_stack with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare k_stacks that are automatically initialized at boot with *K_STACK_DEFINE()*, or to initialize the k_stack in supervisor mode with *k_stack_init()*.

- *k_pipe_alloc_init()* sets up a k_pipe object with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare k_pipes that are automatically initialized at boot with *K_PIPE_DEFINE()*, or to initialize the k_pipe in supervisor mode with *k_pipe_init()*.
- *k_msgq_alloc_init()* sets up a k_msgq object with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare a k_msgq that is automatically initialized at boot with *K_MSGQ_DEFINE()*, or to initialize the k_msgq in supervisor mode with *k_msgq_init()*.
- *k_poll()* when invoked from user mode, needs to make a kernel-side copy of the provided events array while waiting for an event. This copy is freed when *k_poll()* returns for any reason.
- *k_queue_alloc_prepend()* and *k_queue_alloc_append()* allocate a container structure to place the data in, since the internal bookkeeping information that defines the queue cannot be placed in the memory provided by the user.
- *k_object_alloc()* allows for entire kernel objects to be dynamically allocated at runtime and a usable pointer to them returned to the caller.

The relevant API is *k_thread_heap_assign()* which assigns a k_heap to draw these allocations from for the target thread.

If the system heap is enabled, then the system heap may be used with *k_thread_system_pool_assign()*, but it is preferable for different logical applications running on the system to have their own pools.

Memory Domains

The kernel ensures that any user thread will have access to its own stack buffer, plus program text and read-only data. The memory domain APIs are the way to grant access to additional blocks of memory to a user thread.

Conceptually, a memory domain is a collection of some number of memory partitions. The maximum number of memory partitions in a domain is limited by the number of available MPU regions. This is why it is important to minimize the number of boot-time MPU regions.

Memory domains are *not* intended to control access to memory from supervisor mode. In some cases this may be unavoidable; for example some architectures do not allow for the definition of regions which are read-only to user mode but read-write to supervisor mode. A great deal of care must be taken when working with such regions to not unintentionally cause the kernel to crash when accessing such a region. Any attempt to use memory domain APIs to control supervisor mode access is at best undefined behavior; supervisor mode access policy is only intended to be controlled by boot-time memory regions.

Memory domain APIs are only available to supervisor mode. The only control user mode has over memory domains is that any user thread's child threads will automatically become members of the parent's domain.

All threads are members of a memory domain, including supervisor threads (even though this has no implications on their memory access). There is a default domain `k_mem_domain_default` which will be assigned to threads if they have not been specifically assigned to a domain, or inherited a memory domain membership from their parent thread. The main thread starts as a member of the default domain.

Memory Partitions Each memory partition consists of a memory address, a size, and access attributes. It is intended that memory partitions are used to control access to system memory. Defining memory partitions are subject to the following constraints:

- The partition must represent a memory region that can be programmed by the underlying memory management hardware, and needs to conform to any underlying hardware constraints. For example, many MPU-based systems require that partitions be sized to some power of two, and aligned to their own size. For MMU-based systems, the partition must be aligned to a page and the size some multiple of the page size.

- Partitions within the same memory domain may not overlap each other. There is no notion of precedence among partitions within a memory domain. Partitions within a memory domain are assumed to have a higher precedence than any boot-time memory regions, however whether a memory domain partition can overlap a boot-time memory region is architecture specific.
- The same partition may be specified in multiple memory domains. For example there may be a shared memory area that multiple domains grant access to.
- Care must be taken in determining what memory to expose in a partition. It is not appropriate to provide direct user mode access to any memory containing private kernel data.
- Memory domain partitions are intended to control access to system RAM. Configuration of memory partitions which do not correspond to RAM may not be supported by the architecture; this is true for MMU-based systems.

There are two ways to define memory partitions: either manually or automatically.

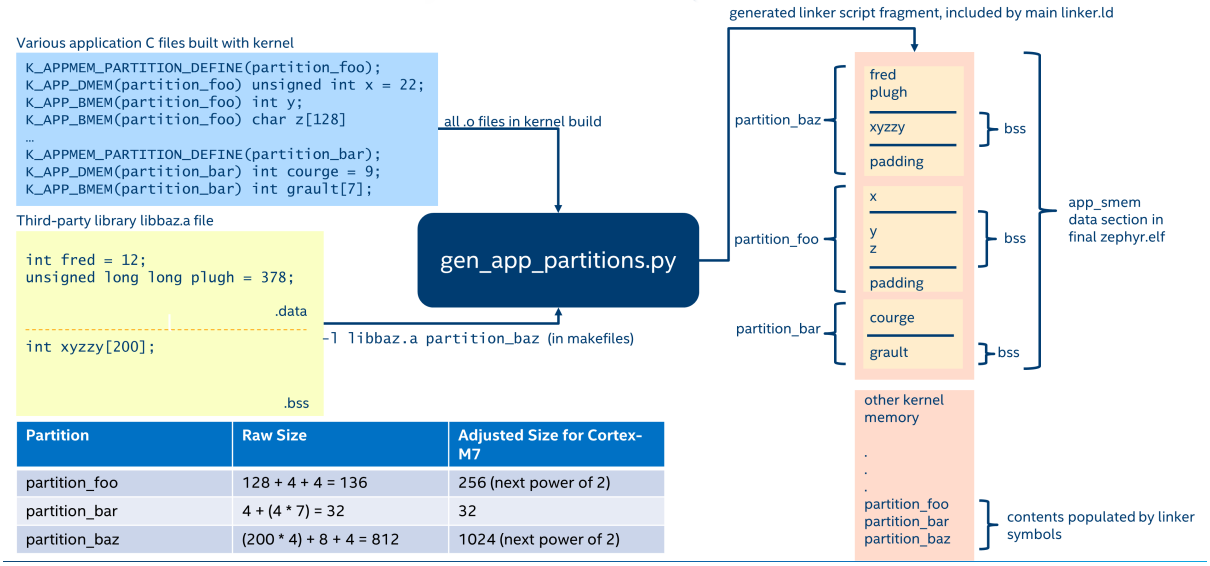
Manual Memory Partitions The following code declares a global array `buf`, and then declares a read-write partition for it which may be added to a domain:

```
uint8_t __aligned(32) buf[32];

K_MEM_PARTITION_DEFINE(my_partition, buf, sizeof(buf),
                       K_MEM_PARTITION_P_RW_U_RW);
```

This does not scale particularly well when we are trying to contain multiple objects spread out across several C files into a single partition.

Automatic Memory Partitions Automatic memory partitions are created by the build system. All globals which need to be placed inside a partition are tagged with their destination partition. The build system will then coalesce all of these into a single contiguous block of memory, zero any BSS variables at boot, and define a memory partition of appropriate base address and size which contains all the tagged data.



Partition	Raw Size	Adjusted Size for Cortex-M7
partition_foo	128 + 4 + 4 = 136	256 (next power of 2)
partition_bar	4 + (4 * 7) = 32	32
partition_baz	(200 * 4) + 8 + 4 = 812	1024 (next power of 2)

Fig. 9: Automatic Memory Domain build flow

Automatic memory partitions are only configured as read-write regions. They are defined with `K_APPMEM_PARTITION_DEFINE()`. Global variables are then routed to this partition using `K_APP_DMEM()` for initialized data and `K_APP_BMEM()` for BSS.

```
#include <app_memory/app_memdomain.h>

/* Declare a k_mem_partition "my_partition" that is read-write to
 * user mode. Note that we do not specify a base address or size.
 */
K_APPMEM_PARTITION_DEFINE(my_partition);

/* The global variable var1 will be inside the bounds of my_partition
 * and be initialized with 37 at boot.
 */
K_APP_DMEM(my_partition) int var1 = 37;

/* The global variable var2 will be inside the bounds of my_partition
 * and be zeroed at boot size K_APP_BMEM() was used, indicating a BSS
 * variable.
 */
K_APP_BMEM(my_partition) int var2;
```

The build system will ensure that the base address of `my_partition` will be properly aligned, and the total size of the region conforms to the memory management hardware requirements, adding padding if necessary.

If multiple partitions are being created, a variadic preprocessor macro can be used as provided in `app_macro_support.h`:

```
FOR_EACH(K_APPMEM_PARTITION_DEFINE, part0, part1, part2);
```

Automatic Partitions for Static Library Globals The build-time logic for setting up automatic memory partitions is in `scripts/gen_app_partitions.py`. If a static library is linked into Zephyr, it is possible to route all the globals in that library to a specific memory partition with the `--library` argument.

For example, if the Newlib C library is enabled, the Newlib globals all need to be placed in `z_libc_partition`. The invocation of the script in the top-level `CMakeLists.txt` adds the following:

```
gen_app_partitions.py ... --library libc.a z_libc_partition ..
```

For pre-compiled libraries there is no support for expressing this in the project-level configuration or build files; the toplevel `CMakeLists.txt` must be edited.

For Zephyr libraries created using `zephyr_library` or `zephyr_library_named` the `zephyr_library_app_memory` function can be used to specify the memory partition where all globals in the library should be placed.

Pre-defined Memory Partitions There are a few memory partitions which are pre-defined by the system:

- `z_malloc_partition` - This partition contains the system-wide pool of memory used by libc malloc(). Due to possible starvation issues, it is not recommended to draw heap memory from a global pool, instead it is better to define various sys_heap objects and assign them to specific memory domains.
- `z_libc_partition` - Contains globals required by the C library and runtime. Required when using either the Minimal C library or the Newlib C Library. Required when option:`CONFIG_STACK_CANARIES` is enabled.

Library-specific partitions are listed in `include/app_memory/partitions.h`. For example, to use the MBEDTLS library from user mode, the `k_mbedtls_partition` must be added to the domain.

Memory Domain Usage

Create a Memory Domain A memory domain is defined using a variable of type *k_mem_domain*. It must then be initialized by calling *k_mem_domain_init()*.

The following code defines and initializes an empty memory domain.

```
struct k_mem_domain app0_domain;

k_mem_domain_init(&app0_domain, 0, NULL);
```

Add Memory Partitions into a Memory Domain There are two ways to add memory partitions into a memory domain.

This first code sample shows how to add memory partitions while creating a memory domain.

```
/* the start address of the MPU region needs to align with its size */
uint8_t __aligned(32) app0_buf[32];
uint8_t __aligned(32) app1_buf[32];

K_MEM_PARTITION_DEFINE(app0_part0, app0_buf, sizeof(app0_buf),
                       K_MEM_PARTITION_P_RW_U_RW);

K_MEM_PARTITION_DEFINE(app0_part1, app1_buf, sizeof(app1_buf),
                       K_MEM_PARTITION_P_RW_U_RO);

struct k_mem_partition *app0_parts[] = {
    app0_part0,
    app0_part1
};

k_mem_domain_init(&app0_domain, ARRAY_SIZE(app0_parts), app0_parts);
```

This second code sample shows how to add memory partitions into an initialized memory domain one by one.

```
/* the start address of the MPU region needs to align with its size */
uint8_t __aligned(32) app0_buf[32];
uint8_t __aligned(32) app1_buf[32];

K_MEM_PARTITION_DEFINE(app0_part0, app0_buf, sizeof(app0_buf),
                       K_MEM_PARTITION_P_RW_U_RW);

K_MEM_PARTITION_DEFINE(app0_part1, app1_buf, sizeof(app1_buf),
                       K_MEM_PARTITION_P_RW_U_RO);

k_mem_domain_add_partition(&app0_domain, &app0_part0);
k_mem_domain_add_partition(&app0_domain, &app0_part1);
```

Note: The maximum number of memory partitions is limited by the maximum number of MPU regions or the maximum number of MMU tables.

Memory Domain Assignment Any thread may join a memory domain, and any memory domain may have multiple threads assigned to it. Threads are assigned to memory domains with an API call:

```
k_mem_domain_add_thread(&app0_domain, app_thread_id);
```

If the thread was already a member of some other domain (including the default domain), it will be removed from it in favor of the new one.

In addition, if a thread is a member of a memory domain, and it creates a child thread, that thread will belong to the domain as well.

Remove a Memory Partition from a Memory Domain The following code shows how to remove a memory partition from a memory domain.

```
k_mem_domain_remove_partition(&app0_domain, &app0_part1);
```

The k_mem_domain_remove_partition() API finds the memory partition that matches the given parameter and removes that partition from the memory domain.

Available Partition Attributes When defining a partition, we need to set access permission attributes to the partition. Since the access control of memory partitions relies on either an MPU or MMU, the available partition attributes would be architecture dependent.

The complete list of available partition attributes for a specific architecture is found in the architecture-specific include file `include/arch/<arch name>/arch.h`, (for example, `include/arch/arm/aarch32/arch.h`.) Some examples of partition attributes are:

```
/* Denote partition is privileged read/write, unprivileged read/write */
K_MEM_PARTITION_P_RW_U_RW
/* Denote partition is privileged read/write, unprivileged read-only */
K_MEM_PARTITION_P_RW_U_RO
```

In almost all cases `K_MEM_PARTITION_P_RW_U_RW` is the right choice.

Configuration Options

Related configuration options:

- `CONFIG_MAX_DOMAIN_PARTITIONS`

API Reference

The following memory domain APIs are provided by include/kernel.h:

group `mem_domain_apis`

Defines

`K_MEM_PARTITION_DEFINE`(name, start, size, attr)

Statically declare a memory partition.

Functions

void `k_mem_domain_init`(struct *k_mem_domain* *domain, uint8_t num_parts, struct *k_mem_partition* *parts[])

Initialize a memory domain.

Initialize a memory domain with given name and memory partitions.

See documentation for *k_mem_domain_add_partition()* for details about partition constraints.

Do not call *k_mem_domain_init()* on the same memory domain more than once, doing so is undefined behavior.

Parameters

- `domain` – The memory domain to be initialized.
- `num_parts` – The number of array items of "parts" parameter.
- `parts` – An array of pointers to the memory partitions. Can be NULL if num_parts is zero.

void `k_mem_domain_add_partition`(struct *k_mem_domain* *domain, struct *k_mem_partition* *part)

Add a memory partition into a memory domain.

Add a memory partition into a memory domain. Partitions must conform to the following constraints:

- Partitions in the same memory domain may not overlap each other.
- Partitions must not be defined which expose private kernel data structures or kernel objects.
- The starting address alignment, and the partition size must conform to the constraints of the underlying memory management hardware, which varies per architecture.
- Memory domain partitions are only intended to control access to memory from user mode threads.
- If CONFIG_EXECUTE_XOR_WRITE is enabled, the partition must not allow both writes and execution.

Violating these constraints may lead to CPU exceptions or undefined behavior.

Parameters

- `domain` – The memory domain to be added a memory partition.
- `part` – The memory partition to be added

void `k_mem_domain_remove_partition`(struct *k_mem_domain* *domain, struct *k_mem_partition* *part)

Remove a memory partition from a memory domain.

Remove a memory partition from a memory domain.

Parameters

- `domain` – The memory domain to be removed a memory partition.
- `part` – The memory partition to be removed

void `k_mem_domain_add_thread`(struct *k_mem_domain* *domain, k_tid_t thread)

Add a thread into a memory domain.

Add a thread into a memory domain. It will be removed from whatever memory domain it previously belonged to.

Parameters

- `domain` – The memory domain that the thread is going to be added into.
- `thread` – ID of thread going to be added into the memory domain.

Variables

struct *k_mem_domain* `k_mem_domain_default`

Default memory domain

All threads are a member of some memory domain, even if running in supervisor mode. Threads belong to this default memory domain if they haven't been added to or inherited membership from some other domain.

This memory domain has the z_libc_partition partition for the C library added to it if exists.

struct `k_mem_partition`

#include <mem_domain.h> Memory Partition.

A memory partition is a region of memory in the linear address space with a specific access policy.

The alignment of the starting address, and the alignment of the size value may have varying requirements based on the capabilities of the underlying memory management hardware; arbitrary values are unlikely to work.

Public Members

uintptr_t `start`

start address of memory partition

size_t `size`

size of memory partition

k_mem_partition_attr_t `attr`

attribute of memory partition

struct `k_mem_domain`

#include <mem_domain.h> Memory Domain.

A memory domain is a collection of memory partitions, used to represent a user thread's access policy for the linear addresss space. A thread may be a member of only one memory domain, but any memory domain may have multiple threads that are members.

Supervisor threads may also be a member of a memory domain; this has no implications on their memory access but can be useful as any child threads inherit the memory domain membership of the parent.

A user thread belonging to a memory domain with no active partitions will have guaranteed access to its own stack buffer, program text, and read-only data.

Public Members

struct *k_mem_partition* `partitions`[CONFIG_MAX_DOMAIN_PARTITIONS]

partitions in the domain

sys_dlist_t `mem_domain_q`

Doubly linked list of member threads

uint8_t num_partitions

number of active partitions in the domain

7.30.3 Kernel Objects

A kernel object can be one of three classes of data:

- A core kernel object, such as a semaphore, thread, pipe, etc.
- A thread stack, which is an array of z_thread_stack_element and declared with *K_THREAD_STACK_DEFINE()*
- A device driver instance (const struct device) that belongs to one of a defined set of subsystems

The set of known kernel objects and driver subsystems is defined in include/kernel.h as k_objects.

Kernel objects are completely opaque to user threads. User threads work with addresses to kernel objects when making API calls, but may never dereference these addresses, doing so will cause a memory protection fault. All kernel objects must be placed in memory that is not accessible by user threads.

Since user threads may not directly manipulate kernel objects, all use of them must go through system calls. In order to perform a system call on a kernel object, checks are performed by system call handler functions that the kernel object address is valid and that the calling thread has sufficient permissions to work with it.

Permission on an object also has the semantics of a reference to an object. This is significant for certain object APIs which do temporary allocations, or objects which themselves have been allocated from a runtime memory pool.

If an object loses all references, two events may happen:

- If the object has an associated cleanup function, the cleanup function may be called to release any runtime-allocated buffers the object was using.
- If the object itself was dynamically allocated, the memory for the object will be freed.

Object Placement

Kernel objects that are only used by supervisor threads have no restrictions and can be located anywhere in the binary, or even declared on stacks. However, to prevent accidental or intentional corruption by user threads, they must not be located in any memory that user threads have direct access to.

In order for a static kernel object to be usable by a user thread via system call APIs, several conditions must be met on how the kernel object is declared:

- The object must be declared as a top-level global at build time, such that it appears in the ELF symbol table. It is permitted to declare kernel objects with static scope. The post-build script *scripts/gen_kobject_list.py* scans the generated ELF file to find kernel objects and places their memory addresses in a special table of kernel object metadata. Kernel objects may be members of arrays or embedded within other data structures.
- Kernel objects must be located in memory reserved for the kernel. They must not be located in any memory partitions that are user-accessible.
- Any memory reserved for a kernel object must be used exclusively for that object. Kernel objects may not be members of a union data type.

Kernel objects that are found but do not meet the above conditions will not be included in the generated table that is used to validate kernel object pointers passed in from user mode.

The debug output of the *scripts/gen_kobject_list.py* script may be useful when debugging why some object was unexpectedly not being tracked. This information will be printed if the script is run with the `--verbose` flag, or if the build system is invoked with verbose output.

Dynamic Objects

Kernel objects may also be allocated at runtime if CONFIG_DYNAMIC_OBJECTS is enabled. In this case, the *k_object_alloc()* API may be used to instantiate an object from the calling thread's resource pool. Such allocations may be freed in two ways:

- Supervisor threads may call *k_object_free()* to force a dynamic object to be released.
- If an object's references drop to zero (which happens when no threads have permissions on it) the object will be automatically freed. User threads may drop their own permission on an object with *k_object_release()*, and their permissions are automatically cleared when a thread terminates. Supervisor threads may additionally revoke references for another thread using *k_object_access_revoke()*.

Because permissions are also used for reference counting, it is important for supervisor threads to acquire permissions on objects they are using even though the access control aspects of the permission system are not enforced.

Implementation Details The *scripts/gen_kobject_list.py* script is a post-build step which finds all the valid kernel object instances in the binary. It accomplishes this by parsing the DWARF debug information present in the generated ELF file for the kernel.

Any instances of structs or arrays corresponding to kernel objects that meet the object placement criteria will have their memory addresses placed in a special perfect hash table of kernel objects generated by the 'gperf' tool. When a system call is made and the kernel is presented with a memory address of what may or may not be a valid kernel object, the address can be validated with a constant-time lookup in this table.

Drivers are a special case. All drivers are instances of *device*, but it is important to know what subsystem a driver belongs to so that incorrect operations, such as calling a UART API on a sensor driver object, can be prevented. When a device struct is found, its API pointer is examined to determine what subsystem the driver belongs to.

The table itself maps kernel object memory addresses to instances of z_object, which has all the metadata for that object. This includes:

- A bitfield indicating permissions on that object. All threads have a numerical ID assigned to them at build time, used to index the permission bitfield for an object to see if that thread has permission on it. The size of this bitfield is controlled by the CONFIG_MAX_THREAD_BYTES option and the build system will generate an error if this value is too low.
- A type field indicating what kind of object this is, which is some instance of k_objects.
- A set of flags for that object. This is currently used to track initialization state and whether an object is public or not.
- An extra data field. The semantics of this field vary by object type, see the definition of z_object_data.

Dynamic objects allocated at runtime are tracked in a runtime red/black tree which is used in parallel to the gperf table when validating object pointers.

Supervisor Thread Access Permission

Supervisor threads can access any kernel object. However, permissions for supervisor threads are still tracked for two reasons:

- If a supervisor thread calls *k_thread_user_mode_enter()*, the thread will then run in user mode with any permissions it had been granted (in many cases, by itself) when it was a supervisor thread.
- If a supervisor thread creates a user thread with the *K_INHERIT_PERMS* option, the child thread will be granted the same permissions as the parent thread, except the parent thread object.

User Thread Access Permission

By default, when a user thread is created, it will only have access permissions on its own thread object. Other kernel objects by default are not usable. Access to them needs to be explicitly or implicitly granted. There are several ways to do this.

- If a thread is created with the *K_INHERIT_PERMS*, that thread will inherit all the permissions of the parent thread, except the parent thread object.
- A thread that has permission on an object, or is running in supervisor mode, may grant permission on that object to another thread via the *k_object_access_grant()* API. The convenience pseudo-function *k_thread_access_grant()* may also be used, which accepts an arbitrary number of pointers to kernel objects and calls *k_object_access_grant()* on each of them. The thread being granted permission, or the object whose access is being granted, do not need to be in an initialized state. If the caller is from user mode, the caller must have permissions on both the kernel object and the target thread object.
- Supervisor threads may declare a particular kernel object to be a public object, usable by all current and future threads with the *k_object_access_all_grant()* API. You must assume that any untrusted or exploited code will then be able to access the object. Use this API with caution!
- If a thread was declared statically with *K_THREAD_DEFINE()*, then the *K_THREAD_ACCESS_GRANT()* may be used to grant that thread access to a set of kernel objects at boot time.

Once a thread has been granted access to an object, such access may be removed with the *k_object_access_revoke()* API. This API is not available to user threads, however user threads may use *k_object_release()* to relinquish their own permissions on an object.

API calls from supervisor mode to set permissions on kernel objects that are not being tracked by the kernel will be no-ops. Doing the same from user mode will result in a fatal error for the calling thread.

Objects allocated with *k_object_alloc()* implicitly grant permission on the allocated object to the calling thread.

Initialization State

Most operations on kernel objects will fail if the object is considered to be in an uninitialized state. The appropriate init function for the object must be performed first.

Some objects will be implicitly initialized at boot:

- Kernel objects that were declared with static initialization macros (such as *K_SEM_DEFINE* for semaphores) will be in an initialized state at build time.
- Device driver objects are considered initialized after their init function is run by the kernel early in the boot process.

If a kernel object is initialized with a private static initializer, the object must have `z_object_init()` called on it at some point by a supervisor thread, otherwise the kernel will consider the object uninitialized if accessed by a user thread. This is very uncommon, typically only for kernel objects that are embedded within some larger struct and initialized statically.

```
struct foo {
    struct k_sem sem;
    ...
};

struct foo my_foo = {
    .sem = Z_SEM_INITIALIZER(my_foo.sem, 0, 1),
    ...
};
```

(continues on next page)

(continued from previous page)

```
...
z_object_init(&my_foo.sem);
...
```

Creating New Kernel Object Types

When implementing new kernel features or driver subsystems, it may be necessary to define some new kernel object types. There are different steps needed for creating core kernel objects and new driver subsystems.

Creating New Core Kernel Objects

- In `scripts/gen_kobject_list.py`, add the name of the struct to the `kobjects` list.

Instances of the new struct should now be tracked.

Creating New Driver Subsystem Kernel Objects All driver instances are *device*. They are differentiated by what API struct they are set to.

- In `scripts/gen_kobject_list.py`, add the name of the API struct for the new subsystem to the `subsystems` list.

Driver instances of the new subsystem should now be tracked.

Configuration Options

Related configuration options:

- `CONFIG_USERSPACE`
- `CONFIG_MAX_THREAD_BYTES`

API Reference

group `usermode_apis`

Defines

`K_THREAD_ACCESS_GRANT(name_, ...)`

Grant a static thread access to a list of kernel objects.

For threads declared with *K_THREAD_DEFINE()*, grant the thread access to a set of kernel objects. These objects do not need to be in an initialized state. The permissions will be granted when the threads are initialized in the early boot sequence.

All arguments beyond the first must be pointers to kernel objects.

Parameters

- `name_` – Name of the thread, as passed to *K_THREAD_DEFINE()*

`K_OBJ_FLAG_INITIALIZED`

Object initialized

K_OBJ_FLAG_PUBLIC

Object is Public

K_OBJ_FLAG_ALLOC

Object allocated

K_OBJ_FLAG_DRIVER

Driver Object

Functions

void k_object_access_grant(const void *object, struct *k_thread* *thread)

Grant a thread access to a kernel object

The thread will be granted access to the object if the caller is from supervisor mode, or the caller is from user mode AND has permissions on both the object and the thread whose access is being granted.

Parameters

- object – Address of kernel object
- thread – Thread to grant access to the object

void k_object_access_revoke(const void *object, struct *k_thread* *thread)

Revoke a thread's access to a kernel object

The thread will lose access to the object if the caller is from supervisor mode, or the caller is from user mode AND has permissions on both the object and the thread whose access is being revoked.

Parameters

- object – Address of kernel object
- thread – Thread to remove access to the object

void k_object_release(const void *object)

Release an object.

Allows user threads to drop their own permission on an object Their permissions are automatically cleared when a thread terminates.

Parameters

- object – The object to be released

void k_object_access_all_grant(const void *object)

Grant all present and future threads access to an object

If the caller is from supervisor mode, or the caller is from user mode and have sufficient permissions on the object, then that object will have permissions granted to it for *all* current and future threads running in the system, effectively becoming a public kernel object.

Use of this API should be avoided on systems that are running untrusted code as it is possible for such code to derive the addresses of kernel objects and perform unwanted operations on them.

It is not possible to revoke permissions on public objects; once public, any thread may use it.

Parameters

- object – Address of kernel object

void *k_object_alloc(enum k_objects otype)

Allocate a kernel object of a designated type

This will instantiate at runtime a kernel object of the specified type, returning a pointer to it. The object will be returned in an uninitialized state, with the calling thread being granted permission on it. The memory for the object will be allocated out of the calling thread's resource pool.

Currently, allocation of thread stacks is not supported.

Parameters

- otype – Requested kernel object type

Returns A pointer to the allocated kernel object, or NULL if memory wasn't available

static inline void k_object_free(void *obj)

Free an object.

Parameters

- obj –

7.30.4 System Calls

User threads run with a reduced set of privileges than supervisor threads: certain CPU instructions may not be used, and they have access to only a limited part of the memory map. System calls (may) allow user threads to perform operations not directly available to them.

When defining system calls, it is very important to ensure that access to the API's private data is done exclusively through system call interfaces. Private kernel data should never be made available to user mode threads directly. For example, the k_queue APIs were intentionally not made available as they store bookkeeping information about the queue directly in the queue buffers which are visible from user mode.

APIs that allow the user to register callback functions that run in supervisor mode should never be exposed as system calls. Reserve these for supervisor-mode access only.

This section describes how to declare new system calls and discusses a few implementation details relevant to them.

Components

All system calls have the following components:

- A **C prototype** prefixed with __syscall for the API. It will be declared in some header under include/ or in another SYSCALL_INCLUDE_DIRS directory. This prototype is never implemented manually, instead it gets created by the *scripts/gen_syscalls.py* script. What gets generated is an inline function which either calls the implementation function directly (if called from supervisor mode) or goes through privilege elevation and validation steps (if called from user mode).
- An **implementation function**, which is the real implementation of the system call. The implementation function may assume that all parameters passed in have been validated if it was invoked from user mode.
- A **verification function**, which wraps the implementation function and does validation of all the arguments passed in.
- An **unmarshalling function**, which is an automatically generated handler that must be included by user source code.

C Prototype

The C prototype represents how the API is invoked from either user or supervisor mode. For example, to initialize a semaphore:

```
__syscall void k_sem_init(struct k_sem *sem, unsigned int initial_count,
                          unsigned int limit);
```

The `__syscall` attribute is very special. To the C compiler, it simply expands to 'static inline'. However to the post-build *scripts/parse_syscalls.py* script, it indicates that this API is a system call. The *scripts/parse_syscalls.py* script does some parsing of the function prototype, to determine the data types of its return value and arguments, and has some limitations:

- Array arguments must be passed in as pointers, not arrays. For example, `int foo[]` or `int foo[12]` is not allowed, but should instead be expressed as `int *foo`.
- Function pointers horribly confuse the limited parser. The workaround is to typedef them first, and then express in the argument list in terms of that typedef.
- `__syscall` must be the first thing in the prototype.

The preprocessor is intentionally not used when determining the set of system calls to generate. However, any generated system calls that don't actually have a verification function defined (because the related feature is not enabled in the kernel configuration) will instead point to a special verification for unimplemented system calls. Data type definitions for APIs should not have conditional visibility to the compiler.

Any header file that declares system calls must include a special generated header at the very bottom of the header file. This header follows the naming convention `syscalls/<name of header file>`. For example, at the bottom of `include/sensor.h`:

```
#include <syscalls/sensor.h>
```

C prototype functions must be declared in one of the directories listed in the CMake variable `SYSCALL_INCLUDE_DIRS`. This list always contains `${ZEPHYR_BASE}/include`, but will also contain `APPLICATION_SOURCE_DIR` when `CONFIG_APPLICATION_DEFINED_SYSCALL` is set, or `${ZEPHYR_BASE}/subsys/testsuite/ztest/include` when `CONFIG_ZTEST` is set. Additional paths can be added to the list through the CMake command line or in CMake code that is run before `${ZEPHYR_BASE}/cmake/app/boilerplate.cmake` is run.

Invocation Context Source code that uses system call APIs can be made more efficient if it is known that all the code inside a particular C file runs exclusively in user mode, or exclusively in supervisor mode. The system will look for the definition of macros `__ZEPHYR_SUPERVISOR__` or `__ZEPHYR_USER__`, typically these will be added to the compiler flags in the build system for the related files.

- If `CONFIG_USERSPACE` is not enabled, all APIs just directly call the implementation function.
- Otherwise, the default case is to make a runtime check to see if the processor is currently running in user mode, and either make the system call or directly call the implementation function as appropriate.
- If `__ZEPHYR_SUPERVISOR__` is defined, then it is assumed that all the code runs in supervisor mode and all APIs just directly call the implementation function. If the code was actually running in user mode, there will be a CPU exception as soon as it tries to do something it isn't allowed to do.
- If `__ZEPHYR_USER__` is defined, then it is assumed that all the code runs in user mode and system calls are unconditionally made.

Implementation Details Declaring an API with `__syscall` causes some code to be generated in C and header files by the *scripts/gen_syscalls.py* script, all of which can be found in the project out directory under `include/generated/`:

- The system call is added to the enumerated type of system call IDs, which is expressed in `include/generated/syscall_list.h`. It is the name of the API in uppercase, prefixed with `K_SYSCALL_`.
- An entry for the system call is created in the dispatch table `_k_syscall_table`, expressed in `include/generated/syscall_dispatch.c`
- A weak verification function is declared, which is just an alias of the 'unimplemented system call' verifier. This is necessary since the real verification function may or may not be built depending on the kernel configuration. For example, if a user thread makes a sensor subsystem API call, but the sensor subsystem is not enabled, the weak verifier will be invoked instead.
- An unmarshalling function is defined in `include/generated/<name>_mrsh.c`

The body of the API is created in the generated system header. Using the example of *k_sem_init()*, this API is declared in `include/kernel.h`. At the bottom of `include/kernel.h` is:

```
#include <syscalls/kernel.h>
```

Inside this header is the body of *k_sem_init()*:

```
static inline void k_sem_init(struct k_sem * sem, unsigned int initial_count,␣
↪unsigned int limit)
{
#ifdef CONFIG_USERSPACE
        if (z_syscall_trap()) {
                arch_syscall_invoke3(*(uintptr_t *)&sem, *(uintptr_t *)&initial_count,
↪ *(uintptr_t *)&limit, K_SYSCALL_K_SEM_INIT);
                return;
        }
        compiler_barrier();
#endif
        z_impl_k_sem_init(sem, initial_count, limit);
}
```

This generates an inline function that takes three arguments with void return value. Depending on context it will either directly call the implementation function or go through a system call elevation. A prototype for the implementation function is also automatically generated.

The final layer is the invocation of the system call itself. All architectures implementing system calls must implement the seven inline functions `_arch_syscall_invoke0()` through `_arch_syscall_invoke6()`. These functions marshal arguments into designated CPU registers and perform the necessary privilege elevation. Parameters of API inline function, before being passed as arguments to system call, are C casted to `uintptr_t` which matches size of register. Exception to above is passing 64-bit parameters on 32-bit systems, in which case 64-bit parameters are split into lower and higher part and passed as two consecutive arguments. There is always a `uintptr_t` type return value, which may be neglected if not needed.

Some system calls may have more than six arguments, but number of arguments passed via registers is limited to six for all architectures. Additional arguments will need to be passed in an array in the source memory space, which needs to be treated as untrusted memory in the verification function. This code (packing, unpacking and validation) is generated automatically as needed in the stub above and in the unmarshalling function.

System calls return `uintptr_t` type value that is C casted, by wrapper, to a return type of API prototype declaration. This means that 64-bit value may not be directly returned, from a system call to its wrapper, on 32-bit systems. To solve the problem the automatically generated wrapper function defines 64-bit intermediate variable, which is considered **untrusted** buffer, on its stack and passes pointer to that variable to the system call, as a final argument. Upon return from the system call the value written to that buffer will be returned by the wrapper function. The problem does not exist on 64-bit systems which are able to return 64-bit values directly.

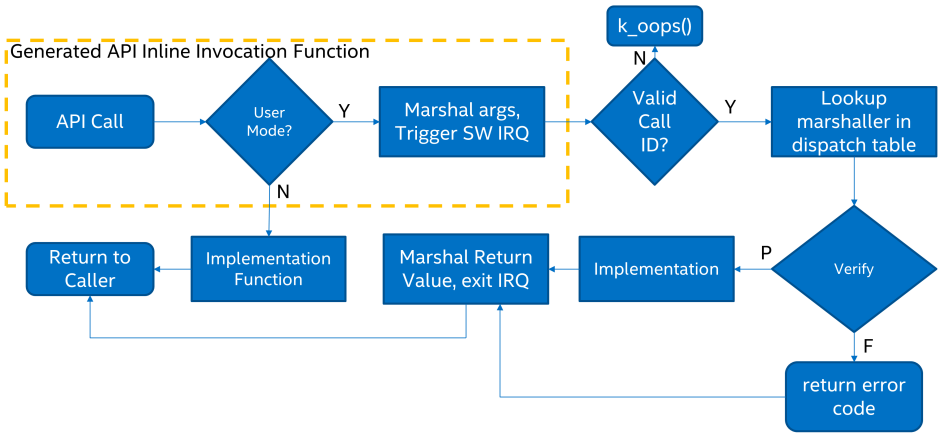


Fig. 10: System Call execution flow

Implementation Function

The implementation function is what actually does the work for the API. Zephyr normally does little to no error checking of arguments, or does this kind of checking with assertions. When writing the implementation function, validation of any parameters is optional and should be done with assertions.

All implementation functions must follow the naming convention, which is the name of the API prefixed with `z_impl_`. Implementation functions may be declared in the same header as the API as a static inline function or declared in some C file. There is no prototype needed for implementation functions, these are automatically generated.

Verification Function

The verification function runs on the kernel side when a user thread makes a system call. When the user thread makes a software interrupt to elevate to supervisor mode, the common system call entry point uses the system call ID provided by the user to look up the appropriate unmarshalling function for that system call and jump into it. This in turn calls the verification function.

Verification and unmarshalling functions only run when system call APIs are invoked from user mode. If an API is invoked from supervisor mode, the implementation is simply called and there is no software trap.

The purpose of the verification function is to validate all the arguments passed in. This includes:

- Any kernel object pointers provided. For example, the semaphore APIs must ensure that the semaphore object passed in is a valid semaphore and that the calling thread has permission on it.
- Any memory buffers passed in from user mode. Checks must be made that the calling thread has read or write permissions on the provided buffer.
- Any other arguments that have a limited range of valid values.

Verification functions involve a great deal of boilerplate code which has been made simpler by some macros in include/syscall_handler.h. Verification functions should be declared using these macros.

Argument Validation Several macros exist to validate arguments:

- `Z_SYSCALL_OBJ()` Checks a memory address to assert that it is a valid kernel object of the expected type, that the calling thread has permissions on it, and that the object is initialized.
- `Z_SYSCALL_OBJ_INIT()` is the same as `Z_SYSCALL_OBJ()`, except that the provided object may be uninitialized. This is useful for verifiers of object init functions.

- `Z_SYSCALL_OBJ_NEVER_INIT()` is the same as `Z_SYSCALL_OBJ()`, except that the provided object must be uninitialized. This is not used very often, currently only for *k_thread_create()*.
- `Z_SYSCALL_MEMORY_READ()` validates a memory buffer of a particular size. The calling thread must have read permissions on the entire buffer.
- `Z_SYSCALL_MEMORY_WRITE()` is the same as `Z_SYSCALL_MEMORY_READ()` but the calling thread must additionally have write permissions.
- `Z_SYSCALL_MEMORY_ARRAY_READ()` validates an array whose total size is expressed as separate arguments for the number of elements and the element size. This macro correctly accounts for multiplication overflow when computing the total size. The calling thread must have read permissions on the total size.
- `Z_SYSCALL_MEMORY_ARRAY_WRITE()` is the same as `Z_SYSCALL_MEMORY_ARRAY_READ()` but the calling thread must additionally have write permissions.
- `Z_SYSCALL_VERIFY_MSG()` does a runtime check of some boolean expression which must evaluate to true otherwise the check will fail. A variant `Z_SYSCALL_VERIFY` exists which does not take a message parameter, instead printing the expression tested if it fails. The latter should only be used for the most obvious of tests.
- `Z_SYSCALL_DRIVER_OP()` checks at runtime if a driver instance is capable of performing a particular operation. While this macro can be used by itself, it's mostly a building block for macros that are automatically generated for every driver subsystem. For instance, to validate the GPIO driver, one could use the `Z_SYSCALL_DRIVER_GPIO()` macro.
- `Z_SYSCALL_SPECIFIC_DRIVER()` is a runtime check to verify that a provided pointer is a valid instance of a specific device driver, that the calling thread has permissions on it, and that the driver has been initialized. It does this by checking the API structure pointer that is stored within the driver instance and ensuring that it matches the provided value, which should be the address of the specific driver's API structure.

If any check fails, the macros will return a nonzero value. The macro `Z_OOPS()` can be used to induce a kernel oops which will kill the calling thread. This is done instead of returning some error condition to keep the APIs the same when calling from supervisor mode.

Verifier Definition All system calls are dispatched to a verifier function with a prefixed `z_vrfy_` name based on the system call. They have exactly the same return type and argument types as the wrapped system call. Their job is to execute the system call (generally by calling the implementation function) after having validated all arguments.

The verifier is itself invoked by an automatically generated unmarshaller function which takes care of unpacking the register arguments from the architecture layer and casting them to the correct type. This is defined in a header file that must be included from user code, generally somewhere after the definition of the verifier in a translation unit (so that it can be inlined).

For example:

```
static int z_vrfy_k_sem_take(struct k_sem *sem, int32_t timeout)
{
    Z_OOPS(Z_SYSCALL_OBJ(sem, K_OBJ_SEM));
    return z_impl_k_sem_take(sem, timeout);
}
#include <syscalls/k_sem_take_mrsh.c>
```

Verification Memory Access Policies Parameters passed to system calls by reference require special handling, because the value of these parameters can be changed at any time by any user thread that has access to the memory that parameter points to. If the kernel makes any logical decisions based on the contents of this memory, this can open up the kernel to attacks even if checking is done. This is a class of exploits known as TOCTOU (Time Of Check to Time Of Use).

The proper procedure to mitigate these attacks is to make a copies in the verification function, and only perform parameter checks on the copies, which user threads will never have access to. The implementation functions get passed the copy and not the original data sent by the user. The `z_user_to_copy()` and `z_user_from_copy()` APIs exist for this purpose.

There is one exception in place, with respect to large data buffers which are only used to provide a memory area that is either only written to, or whose contents are never used for any validation or control flow. Further discussion of this later in this section.

As a first example, consider a parameter which is used as an output parameter for some integral value:

```
int z_vrfy_some_syscall(int *out_param)
{
    int local_out_param;
    int ret;

    ret = z_impl_some_syscall(&local_out_param);
    Z_OOPS(z_user_to_copy(out_param, &local_out_param, sizeof(*out_param)));
    return ret;
}
```

Here we have allocated `local_out_param` on the stack, passed its address to the implementation function, and then used `z_user_to_copy()` to fill in the memory passed in by the caller.

It might be tempting to do something more concise:

```
int z_vrfy_some_syscall(int *out_param)
{
    Z_OOPS(Z_SYSCALL_MEMORY_WRITE(out_param, sizeof(*out_param)));
    return z_impl_some_syscall(out_param);
}
```

However, this is unsafe if the implementation ever does any reads to this memory as part of its logic. For example, it could be used to store some counter value, and this could be meddled with by user threads that have access to its memory. It is by far safest for small integral values to do the copying as shown in the first example.

Some parameters may be input/output. For instance, it's not uncommon to see APIs which pass in a pointer to some `size_t` which is a maximum allowable size, which is then updated by the implementation to reflect the actual number of bytes processed. This too should use a stack copy:

```
int z_vrfy_in_out_syscall(size_t *size_ptr)
{
    size_t size;
    int ret;

    Z_OOPS(z_user_from_copy(&size, size_ptr, sizeof(size));
    ret = z_impl_in_out_syscall(&size);
    *size_ptr = size;
    return ret;
}
```

Many system calls pass in structures or even linked data structures. All should be copied. Typically this is done by allocating copies on the stack:

```
struct bar {
    ...
};

struct foo {
```

(continues on next page)

(continued from previous page)

```
    ...
    struct bar *bar_left;
    struct bar *bar_right;
};

int z_vrfy_must_alloc(struct foo *foo)
{
    int ret;
    struct foo foo_copy;
    struct bar bar_right_copy;
    struct bar bar_left_copy;

    Z_OOPS(z_user_from_copy(&foo_copy, foo, sizeof(*foo)));
    Z_OOPS(z_user_from_copy(&bar_right_copy, foo_copy.bar_right,
                            sizeof(struct bar)));
    foo_copy.bar_right = &bar_right_copy;
    Z_OOPS(z_user_from_copy(&bar_left_copy, foo_copy.bar_left,
                            sizeof(struct bar)));
    foo_copy.bar_left = &bar_left_copy;

    return z_impl_must_alloc(&foo_copy);
}
```

In some cases the amount of data isn't known at compile time or may be too large to allocate on the stack. In this scenario, it may be necessary to draw memory from the caller's resource pool via `z_thread_malloc()`. This should always be considered last resort. Functional safety programming guidelines heavily discourage usage of heap and the fact that a resource pool is used must be clearly documented. Any issues with allocation must be reported, to a caller, with returning the `-ENOMEM` . The `Z_OOPS()` should never be used to verify if resource allocation has been successful.

```
struct bar {
    ...
};

struct foo {
    size_t count;
    struct bar *bar_list; /* array of struct bar of size count */
};

int z_vrfy_must_alloc(struct foo *foo)
{
    int ret;
    struct foo foo_copy;
    struct bar *bar_list_copy;
    size_t bar_list_bytes;

    /* Safely copy foo into foo_copy */
    Z_OOPS(z_user_from_copy(&foo_copy, foo, sizeof(*foo)));

    /* Bounds check the count member, in the copy we made */
    if (foo_copy.count > 32) {
        return -EINVAL;
    }

    /* Allocate RAM for the bar_list, replace the pointer in
     * foo_copy */
```

(continues on next page)

(continued from previous page)

```
    bar_list_bytes = foo_copy.count * sizeof(struct_bar);
    bar_list_copy = z_thread_malloc(bar_list_bytes);
    if (bar_list_copy == NULL) {
        return -ENOMEM;
    }
    Z_OOPS(z_user_from_copy(bar_list_copy, foo_copy.bar_list,
                            bar_list_bytes));
    foo_copy.bar_list = bar_list_copy;

    ret = z_impl_must_alloc(&foo_copy);

    /* All done with the memory, free it and return */
    k_free(foo_copy.bar_list_copy);
    return ret;
}
```

Finally, we must consider large data buffers. These represent areas of user memory which either have data copied out of, or copied into. It is permitted to pass these pointers to the implementation function directly. The caller's access to the buffer still must be validated with `Z_SYSCALL_MEMORY` APIs. The following constraints need to be met:

- If the buffer is used by the implementation function to write data, such as data captured from some MMIO region, the implementation function must only write this data, and never read it.
- If the buffer is used by the implementation function to read data, such as a block of memory to write to some hardware destination, this data must be read without any processing. No conditional logic can be implemented due to the data buffer's contents. If such logic is required a copy must be made.
- The buffer must only be used synchronously with the call. The implementation must not ever save the buffer address and use it asynchronously, such as when an interrupt fires.

```
int z_vrfy_get_data_from_kernel(void *buf, size_t size)
{
    Z_OOPS(Z_SYSCALL_MEMORY_WRITE(buf, size));
    return z_impl_get_data_from_kernel(buf, size);
}
```

Verification Return Value Policies When verifying system calls, it's important to note which kinds of verification failures should propagate a return value to the caller, and which should simply invoke `Z_OOPS()` which kills the calling thread. The current conventions are as follows:

1. For system calls that are defined but not compiled, invocations of these missing system calls are routed to `handler_no_syscall()` which invokes `Z_OOPS()`.
2. Any invalid access to memory found by the set of `Z_SYSCALL_MEMORY` APIs, `z_user_from_copy()`, `z_user_to_copy()` should trigger a `Z_OOPS`. This happens when the caller doesn't have appropriate permissions on the memory buffer or some size calculation overflowed.
3. Most system calls take kernel object pointers as an argument, checked either with one of the `Z_SYSCALL_OBJ` functions, `Z_SYSCALL_DRIVER_nnnnn`, or manually using `z_object_validate()`. These can fail for a variety of reasons: missing driver API, bad kernel object pointer, wrong kernel object type, or improper initialization state. These issues should always invoke `Z_OOPS()`.
4. Any error resulting from a failed memory heap allocation, often from invoking `z_thread_malloc()`, should propagate `-ENOMEM` to the caller.
5. General parameter checks should be done in the implementation function, in most cases using `CHECKIF()`.

- The behavior of `CHECKIF()` depends on the kernel configuration, but if user mode is enabled, `CONFIG_RUNTIME_ERROR_CHECKS` is enforced, which guarantees that these checks will be made and a return value propagated.

6. It is totally forbidden for any kind of kernel mode callback function to be registered from user mode. APIs which simply install callbacks shall not be exposed as system calls. Some driver subsystem APIs may take optional function callback pointers. User mode verification functions for these APIs must enforce that these are NULL and should invoke `Z_OOPS()` if not.
7. Some parameter checks are enforced only from user mode. These should be checked in the verification function and propagate a return value to the caller if possible.

There are some known exceptions to these policies currently in Zephyr:

- *k_thread_join()* and *k_thread_abort()* are no-ops if the thread object isn't initialized. This is because for threads, the initialization bit pulls double-duty to indicate whether a thread is running, cleared upon exit. See #23030.
- *k_thread_create()* invokes `Z_OOPS()` for parameter checks, due to a great deal of existing code ignoring the return value. This will also be addressed by #23030.
- *k_thread_abort()* invokes `Z_OOPS()` if an essential thread is aborted, as the function has no return value.
- Various system calls related to logging invoke `Z_OOPS()` when bad parameters are passed in as they do not propagate errors.

Configuration Options

Related configuration options:

- `CONFIG_USERSPACE`

APIs

Helper macros for creating system call verification functions are provided in include/syscall_handler.h:

- `Z_SYSCALL_OBJ()`
- `Z_SYSCALL_OBJ_INIT()`
- `Z_SYSCALL_OBJ_NEVER_INIT()`
- `Z_OOPS()`
- `Z_SYSCALL_MEMORY_READ()`
- `Z_SYSCALL_MEMORY_WRITE()`
- `Z_SYSCALL_MEMORY_ARRAY_READ()`
- `Z_SYSCALL_MEMORY_ARRAY_WRITE()`
- `Z_SYSCALL_VERIFY_MSG()`
- `Z_SYSCALL_VERIFY`

Functions for invoking system calls are defined in include/syscall.h:

- `_arch_syscall_invoke0()`
- `_arch_syscall_invoke1()`
- `_arch_syscall_invoke2()`
- `_arch_syscall_invoke3()`
- `_arch_syscall_invoke4()`

- `_arch_syscall_invoke5()`
- `_arch_syscall_invoke6()`

7.30.5 MPU Stack Objects

Thread Stack Creation

Thread stacks are declared statically with *K_THREAD_STACK_DEFINE()* or embedded within structures using *K_THREAD_STACK_MEMBER()*

For architectures which utilize memory protection unit (MPU) hardware, stacks are physically contiguous allocations. This contiguous allocation has implications for the placement of stacks in memory, as well as the implementation of other features such as stack protection and userspace. The implications for placement are directly attributed to the alignment requirements for MPU regions. This is discussed in the memory placement section below.

Stack Guards

Stack protection mechanisms require hardware support that can restrict access to memory. Memory protection units can provide this kind of support. The MPU provides a fixed number of regions. Each region contains information about the start, end, size, and access attributes to be enforced on that particular region.

Stack guards are implemented by using a single MPU region and setting the attributes for that region to not allow write access. If invalid accesses occur, a fault ensues. The stack guard is defined at the bottom (the lowest address) of the stack.

Memory Placement

During stack creation, a set of constraints are enforced on the allocation of memory. These constraints include determining the alignment of the stack and the correct sizing of the stack. During linking of the binary, these constraints are used to place the stacks properly.

The main source of the memory constraints is the MPU design for the SoC. The MPU design may require specific constraints on the region definition. These can include alignment of beginning and end addresses, sizes of allocations, or even interactions between overlapping regions.

Some MPUs require that each region be aligned to a power of two. These SoCs will have `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT` defined. This means that a 1500 byte stack should be aligned to a 2kB boundary and the stack size should also be adjusted to 2kB to ensure that nothing else is placed in the remainder of the region. SoCs which include the unmodified ARM v7m MPU will have these constraints.

Some ARM MPUs use start and end addresses to define MPU regions and both the start and end addresses require 32 byte alignment. An example of this kind of MPU is found in the NXP FRDM K64F.

MPUs may have a region priority mechanisms that use the highest priority region that covers the memory access to determine the enforcement policy. Others may logically OR regions to determine enforcement policy.

Size and alignment constraints may result in stack allocations being larger than the requested size. Region priority mechanisms may result in some added complexity when implementing stack guards.

7.30.6 MPU Backed Userspace

The MPU backed userspace implementation requires the creation of a secondary set of stacks. These stacks exist in a 1:1 relationship with each thread stack defined in the system. The privileged stacks are created as a part of the build process.

A post-build script *scripts/gen_kobject_list.py* scans the generated ELF file and finds all of the thread stack objects. A set of privileged stacks, a lookup table, and a set of helper functions are created and added to the image.

During the process of dropping a thread to user mode, the privileged stack information is filled in and later used by the swap and system call infrastructure to configure the MPU regions properly for the thread stack and guard (if applicable).

During system calls, the user mode thread's access to the system call and the passed-in parameters are all validated. The user mode thread is then elevated to privileged mode, the stack is switched to use the privileged stack, and the call is made to the specified kernel API. On return from the kernel API, the thread is set back to user mode and the stack is restored to the user stack.

7.31 Utilities

This page contains reference documentation for `<sys/util.h>`, which provides miscellaneous utility functions and macros.

group `sys-util`

Defines

`POINTER_TO_UINT(x)`
: Cast `x`, a pointer, to an unsigned integer.

`UINT_TO_POINTER(x)`
: Cast `x`, an unsigned integer, to a `void*`.

`POINTER_TO_INT(x)`
: Cast `x`, a pointer, to a signed integer.

`INT_TO_POINTER(x)`
: Cast `x`, a signed integer, to a `void*`.

`BITS_PER_LONG`
: Number of bits in a long int.

`GENMASK(h, l)`
: Create a contiguous bitmask starting at bit position `l` and ending at position `h`.

`ZERO_OR_COMPILE_ERROR(cond)`
: 0 if `cond` is true-ish; causes a compile error otherwise.

`IS_ARRAY(array)`
: Zero if `array` has an array type, a compile error otherwise.

  This macro is available only from C, not C++.

ARRAY_SIZE(array)

Number of elements in the given array.

In C++, due to language limitations, this will accept as array any type that implements operator[]. The results may not be particulary meaningful in this case.

In C, passing a pointer as array causes a compile error.

PART_OF_ARRAY(array, ptr)

Check if a pointer ptr lies within array.

In C but not C++, this causes a compile error if array is not an array (e.g. if ptr and array are mixed up).

Parameters

- ptr – a pointer
- array – an array

Returns 1 if ptr is part of array, 0 otherwise

CONTAINER_OF(ptr, type, field)

Get a pointer to a container structure from an element.

Example:

```
struct foo {
   int bar;
};

struct foo my_foo;
int *ptr = &my_foo.bar;

struct foo *container = CONTAINER_OF(ptr, struct foo, bar);
```

Above, container points at my_foo.

Parameters

- ptr – pointer to a structure element
- type – name of the type that ptr is an element of
- field – the name of the field within the struct ptr points to

Returns a pointer to the structure that contains ptr

ROUND_UP(x, align)

Value of x rounded up to the next multiple of align, which must be a power of 2.

ROUND_DOWN(x, align)

Value of x rounded down to the previous multiple of align, which must be a power of 2.

WB_UP(x)

Value of x rounded up to the next word boundary.

WB_DN(x)

Value of x rounded down to the previous word boundary.

ceiling_fraction(numerator, divider)

Ceiling function applied to numerator / divider as a fraction.

MAX(a, b)

The larger value between a and b.

Note: Arguments are evaluated twice.

MIN(a, b)

The smaller value between a and b.

Note: Arguments are evaluated twice.

CLAMP(val, low, high)

Clamp a value to a given range.

Note: Arguments are evaluated multiple times.

KB(x)

Number of bytes in x kibibytes.

MB(x)

Number of bytes in x mebibytes.

GB(x)

Number of bytes in x gibibytes.

KHZ(x)

Number of Hz in x kHz.

MHZ(x)

Number of Hz in x MHz.

BIT(n)

Unsigned integer with bit position n set (signed in assembly language).

BIT64(_n)

64-bit unsigned integer with bit position _n set.

WRITE_BIT(var, bit, set)

Set or clear a bit depending on a boolean value.

The argument var is a variable whose value is written to as a side effect.

Parameters

- var – Variable to be altered
- bit – Bit number
- set – if 0, clears bit in var; any other value sets bit

BIT_MASK(n)

Bit mask with bits 0 through n-1 (inclusive) set, or 0 if n is 0.

BIT64_MASK(n)

64-bit bit mask with bits 0 through n-1 (inclusive) set, or 0 if n is 0.

IS_ENABLED(config_macro)

Check for macro definition in compiler-visible expressions.

This trick was pioneered in Linux as the config_enabled() macro. It has the effect of taking a macro value that may be defined to "1" or may not be defined at all and turning it into a

literal expression that can be handled by the C compiler instead of just the preprocessor. It is often used with a CONFIG_FOO macro which may be defined to 1 via Kconfig, or left undefined.

That is, it works similarly to #if defined(CONFIG_FOO) except that its expansion is a C expression. Thus, much #ifdef usage can be replaced with equivalents like:

```
if (IS_ENABLED(CONFIG_FOO)) {
        do_something_with_foo
}
```

This is cleaner since the compiler can generate errors and warnings for do_something_with_foo even when CONFIG_FOO is undefined.

Parameters

- config_macro – Macro to check

Returns 1 if config_macro is defined to 1, 0 otherwise (including if config_macro is not defined)

COND_CODE_1(_flag, _if_1_code, _else_code)

Insert code depending on whether _flag expands to 1 or not.

This relies on similar tricks as *IS_ENABLED()*, but as the result of _flag expansion, results in either _if_1_code or _else_code is expanded.

To prevent the preprocessor from treating commas as argument separators, the _if_1_code and _else_code expressions must be inside brackets/parentheses: (). These are stripped away during macro expansion.

Example:

```
COND_CODE_1(CONFIG_FLAG, (uint32_t x;), (there_is_no_flag();))
```

If CONFIG_FLAG is defined to 1, this expands to:

```
uint32_t x;
```

It expands to there_is_no_flag(); otherwise.

This could be used as an alternative to:

```
#if defined(CONFIG_FLAG) && (CONFIG_FLAG == 1)
#define MAYBE_DECLARE(x) uint32_t x
#else
#define MAYBE_DECLARE(x) there_is_no_flag()
#endif

MAYBE_DECLARE(x);
```

However, the advantage of *COND_CODE_1()* is that code is resolved in place where it is used, while the #if method defines MAYBE_DECLARE on two lines and requires it to be invoked again on a separate line. This makes *COND_CODE_1()* more concise and also sometimes more useful when used within another macro's expansion.

Note: _flag can be the result of preprocessor expansion, e.g. an expression involving *NUM_VA_ARGS_LESS_1(...)*. However, _if_1_code is only expanded if _flag expands to the integer literal 1. Integer expressions that evaluate to 1, e.g. after doing some arithmetic, will not work.

Parameters

- _flag – evaluated flag
- _if_1_code – result if _flag expands to 1; must be in parentheses
- _else_code – result otherwise; must be in parentheses

COND_CODE_0(_flag, _if_0_code, _else_code)

Like *COND_CODE_1()* except tests if _flag is 0.

This is like *COND_CODE_1()*, except that it tests whether _flag expands to the integer literal 0. It expands to _if_0_code if so, and _else_code otherwise; both of these must be enclosed in parentheses.

See also:

COND_CODE_1()

Parameters

- _flag – evaluated flag
- _if_0_code – result if _flag expands to 0; must be in parentheses
- _else_code – result otherwise; must be in parentheses

IF_ENABLED(_flag, _code)

Insert code if _flag is defined and equals 1.

Like *COND_CODE_1()*, this expands to _code if _flag is defined to 1; it expands to nothing otherwise.

Example:

```
IF_ENABLED(CONFIG_FLAG, (uint32_t foo;))
```

If CONFIG_FLAG is defined to 1, this expands to:

```
uint32_t foo;
```

and to nothing otherwise.

It can be considered as a more compact alternative to:

```
#if defined(CONFIG_FLAG) && (CONFIG_FLAG == 1)
uint32_t foo;
#endif
```

Parameters

- _flag – evaluated flag
- _code – result if _flag expands to 1; must be in parentheses

IS_EMPTY(a)

Check if a macro has a replacement expression.

If a is a macro defined to a nonempty value, this will return true, otherwise it will return false. It only works with defined macros, so an additional #ifdef test may be needed in some cases.

This macro may be used with *COND_CODE_1()* and *COND_CODE_0()* while processing to avoid processing empty arguments.

Note that this macro is intended to check macro names that evaluate to replacement lists being empty or containing numbers or macro name like tokens.

Example:

```
#define EMPTY
#define NON_EMPTY  1
#undef  UNDEFINED
IS_EMPTY(EMPTY)
IS_EMPTY(NON_EMPTY)
IS_EMPTY(UNDEFINED)
#if defined(EMPTY) && IS_EMPTY(EMPTY) == true
some_conditional_code
#endif
```

In above examples, the invocations of *IS_EMPTY(...)* return `true`, `false`, and `true`; `some_conditional_code` is included.

Note: Not all arguments are accepted by this macro and compilation will fail if argument cannot be concatenated with literal constant. That will happen if argument does not start with letter or number. Example arguments that will fail during compilation: .arg, (arg), "arg", {arg}.

Parameters

- `a` – macro to check for emptiness

`LIST_DROP_EMPTY(...)`

Remove empty arguments from list.

During macro expansion, and other preprocessor generated lists may contain empty elements, e.g.:

```
#define LIST ,a,b,,d,
```

Using EMPTY to show each empty element, LIST contains:

```
EMPTY, a, b, EMPTY, d
```

When processing such lists, e.g. using *FOR_EACH()*, all empty elements will be processed, and may require filtering out. To make that process easier, it is enough to invoke LIST_DROP_EMPTY which will remove all empty elements.

Example:

```
LIST_DROP_EMPTY(LIST)
```

expands to:

```
a, b, d
```

Parameters

- `...` – list to be processed

`EMPTY`

Macro with an empty expansion.

This trivial definition is provided for readability when a macro should expand to an empty result, which e.g. is sometimes needed to silence checkpatch.

Example:

```
#define LIST_ITEM(n) , item##n
```

The above would cause checkpatch to complain, but:

```
#define LIST_ITEM(n) EMPTY, item##n
```

would not.

IDENTITY(V)

Macro that expands to its argument.

This is useful in macros like *FOR_EACH()* when there is no transformation required on the list elements.

Parameters

- V – any value

GET_ARG_N(N, ...)

Get nth argument from argument list.

Parameters

- N – Argument index to fetch. Counter from 1.
- ... – Variable list of argments from which one argument is returned.

Returns Nth argument.

GET_ARGS_LESS_N(N, ...)

Strips n first arguments from the argument list.

Parameters

- N – Number of arguments to discard.
- ... – Variable list of argments.

Returns argument list without N first arguments.

UTIL_OR(a, b)

Like `a || b`, but does evaluation and short-circuiting at C preprocessor time.

This is not the same as the binary `||` operator; in particular, `a` should expand to an integer literal 0 or 1. However, `b` can be any value.

This can be useful when `b` is an expression that would cause a build error when `a` is 1.

UTIL_AND(a, b)

Like `a && b`, but does evaluation and short-circuiting at C preprocessor time.

This is not the same as the binary `&&`, however; in particular, `a` should expand to an integer literal 0 or 1. However, `b` can be any value.

This can be useful when `b` is an expression that would cause a build error when `a` is 0.

UTIL_LISTIFY(LEN, F, ...)

Generates a sequence of code.

Example:

```
#define FOO(i, _) MY_PWM ## i ,
{ UTIL_LISTIFY(PWM_COUNT, FOO) }
```

The above two lines expand to:

{ MY_PWM0 , MY_PWM1 , }

Note: Calling UTIL_LISTIFY with undefined arguments has undefined behavior.

Parameters

- `LEN` – The length of the sequence. Must be an integer literal less than 255.
- `F` – A macro function that accepts at least two arguments: `F(i, ...)`. `F` is called repeatedly in the expansion. Its first argument `i` is the index in the sequence, and the variable list of arguments passed to UTIL_LISTIFY are passed through to `F`.

`FOR_EACH(F, sep, ...)`

Call a macro `F` on each provided argument with a given separator between each call.

Example:

```
#define F(x) int a##x
FOR_EACH(F, (;), 4, 5, 6);
```

This expands to:

```
int a4;
int a5;
int a6;
```

Parameters

- `F` – Macro to invoke
- `sep` – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – Variable argument list. The macro `F` is invoked as `F(element)` for each element in the list.

`FOR_EACH_NONEMPTY_TERM(F, term, ...)`

Like *FOR_EACH()*, but with a terminator instead of a separator, and drops empty elements from the argument list.

The `sep` argument to *FOR_EACH(F, (sep), a, b)* is a separator which is placed between calls to `F`, like this:

```
FOR_EACH(F, (sep), a, b) // F(a) sep F(b)
                         //              ^^^ no sep here!
```

By contrast, the `term` argument to *FOR_EACH_NONEMPTY_TERM(F, (term),a, b)* is added after each time `F` appears in the expansion:

```
FOR_EACH_NONEMPTY_TERM(F, (term), a, b) // F(a) term F(b) term
                                        //               ^^^^
```

Further, any empty elements are dropped:

```
FOR_EACH_NONEMPTY_TERM(F, (term), a, EMPTY, b) // F(a) term F(b) term
```

This is more convenient in some cases, because *FOR_EACH_NONEMPTY_TERM()* expands to nothing when given an empty argument list, and it's often cumbersome to write a macro `F` that does the right thing even when given an empty argument.

One example is when may or may not be empty, and the results are embedded in a larger initializer:

```
#define SQUARE(x) ((x)*(x))

int my_array[] = {
        FOR_EACH_NONEMPTY_TERM(SQUARE, (,), FOO(...))
        FOR_EACH_NONEMPTY_TERM(SQUARE, (,), BAR(...))
        FOR_EACH_NONEMPTY_TERM(SQUARE, (,), BAZ(...))
};
```

This is more convenient than:

a. figuring out whether the `FOO`, `BAR`, and `BAZ` expansions are empty and adding a comma manually (or not) between *FOR_EACH()* calls

b. rewriting SQUARE so it reacts appropriately when "x" is empty (which would be necessary if e.g. `FOO` expands to nothing)

Parameters

- `F` – Macro to invoke on each nonempty element of the variable arguments
- `term` – Terminator (e.g. comma or semicolon) placed after each invocation of F. Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – Variable argument list. The macro `F` is invoked as `F(element)` for each nonempty element in the list.

`FOR_EACH_IDX(F, sep, ...)`

Call macro `F` on each provided argument, with the argument's index as an additional parameter.

This is like *FOR_EACH()*, except `F` should be a macro which takes two arguments: `F(index, variable_arg)`.

Example:

```
#define F(idx, x) int a##idx = x
FOR_EACH_IDX(F, (;), 4, 5, 6);
```

This expands to:

```
int a0 = 4;
int a1 = 5;
int a2 = 6;
```

Parameters

- `F` – Macro to invoke
- `sep` – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- `...` – Variable argument list. The macro `F` is invoked as `F(index, element)` for each element in the list.

FOR_EACH_FIXED_ARG(F, sep, fixed_arg, ...)

Call macro F on each provided argument, with an additional fixed argument as a parameter.

This is like *FOR_EACH()*, except F should be a macro which takes two arguments: F(variable_arg, fixed_arg).

Example:

```
static void func(int val, void *dev);
FOR_EACH_FIXED_ARG(func, (;), dev, 4, 5, 6);
```

This expands to:

```
func(4, dev);
func(5, dev);
func(6, dev);
```

Parameters

- F – Macro to invoke
- sep – Separator (e.g. comma or semicolon). Must be in parentheses; this is required to enable providing a comma as separator.
- fixed_arg – Fixed argument passed to F as the second macro parameter.
- ... – Variable argument list. The macro F is invoked as F(element, fixed_arg) for each element in the list.

FOR_EACH_IDX_FIXED_ARG(F, sep, fixed_arg, ...)

Calls macro F for each variable argument with an index and fixed argument.

This is like the combination of *FOR_EACH_IDX()* with *FOR_EACH_FIXED_ARG()*.

Example:

```
#define F(idx, x, fixed_arg) int fixed_arg##idx = x
FOR_EACH_IDX_FIXED_ARG(F, (;), a, 4, 5, 6);
```

This expands to:

```
int a0 = 4;
int a1 = 5;
int a2 = 6;
```

Parameters

- F – Macro to invoke
- sep – Separator (e.g. comma or semicolon). Must be in parentheses; This is required to enable providing a comma as separator.
- fixed_arg – Fixed argument passed to F as the third macro parameter.
- ... – Variable list of arguments. The macro F is invoked as F(index, element, fixed_arg) for each element in the list.

REVERSE_ARGS(...)

Reverse arguments order.

Parameters

- ... – Variable argument list.

NUM_VA_ARGS_LESS_1(...)

Number of arguments in the variable arguments list minus one.

Parameters

- ... – List of arguments

Returns Number of variadic arguments in the argument list, minus one

MACRO_MAP_CAT(...)

Mapping macro that pastes results together.

This is similar to *FOR_EACH()* in that it invokes a macro repeatedly on each element of . However, unlike *FOR_EACH()*, *MACRO_MAP_CAT()* pastes the results together into a single token.

For example, with this macro FOO:

```
#define FOO(x) item_##x##_
```

MACRO_MAP_CAT(FOO, a, b, c) , expands to the token:

```
item_a_item_b_item_c_
```

Parameters

- ... – Macro to expand on each argument, followed by its arguments. (The macro should take exactly one argument.)

Returns The results of expanding the macro on each argument, all pasted together

MACRO_MAP_CAT_N(N, ...)

Mapping macro that pastes a fixed number of results together.

Similar to *MACRO_MAP_CAT()*, but expects a fixed number of arguments. If more arguments are given than are expected, the rest are ignored.

Parameters

- N – Number of arguments to map
- ... – Macro to expand on each argument, followed by its arguments. (The macro should take exactly one argument.)

Returns The results of expanding the macro on each argument, all pasted together

Functions

static inline bool is_power_of_two(unsigned int x)

Is x a power of two?

Parameters

- x – value to check

Returns true if x is a power of two, false otherwise

static inline int64_t arithmetic_shift_right(int64_t value, uint8_t shift)

Arithmetic shift right.

Parameters

- value – value to shift
- shift – number of bits to shift

Returns `value` shifted right by `shift`; opened bit positions are filled with the sign bit

static inline void `bytecpy`(void *dst, const void *src, size_t size)

byte by byte memcpy.

Copy `size` bytes of `src` into `dest`. This is guaranteed to be done byte by byte.

Parameters

- `dst` – Pointer to the destination memory.
- `src` – Pointer to the source of the data.
- `size` – The number of bytes to copy.

static inline void `byteswp`(void *a, void *b, size_t size)

byte by byte swap.

Swap *size* bytes between memory regions *a* and *b*. This is guaranteed to be done byte by byte.

Parameters

- `a` – Pointer to the the first memory region.
- `b` – Pointer to the the second memory region.
- `size` – The number of bytes to swap.

int `char2hex`(char c, uint8_t *x)

Convert a single character into a hexadecimal nibble.

Parameters

- `c` – The character to convert
- `x` – The address of storage for the converted number.

Returns Zero on success or (negative) error code otherwise.

int `hex2char`(uint8_t x, char *c)

Convert a single hexadecimal nibble into a character.

Parameters

- `c` – The number to convert
- `x` – The address of storage for the converted character.

Returns Zero on success or (negative) error code otherwise.

size_t `bin2hex`(const uint8_t *buf, size_t buflen, char *hex, size_t hexlen)

Convert a binary array into string representation.

Parameters

- `buf` – The binary array to convert
- `buflen` – The length of the binary array to convert
- `hex` – Address of where to store the string representation.
- `hexlen` – Size of the storage area for string representation.

Returns The length of the converted string, or 0 if an error occurred.

size_t `hex2bin`(const char *hex, size_t hexlen, uint8_t *buf, size_t buflen)

Convert a hexadecimal string into a binary array.

Parameters

- `hex` – The hexadecimal string to convert

- `hexlen` – The length of the hexadecimal string to convert.
- `buf` – Address of where to store the binary data
- `buflen` – Size of the storage area for binary data

Returns The length of the binary array, or 0 if an error occurred.

static inline uint8_t `bcd2bin(uint8_t bcd)`

Convert a binary coded decimal (BCD 8421) value to binary.

Parameters

- `bcd` – BCD 8421 value to convert.

Returns Binary representation of input value.

static inline uint8_t `bin2bcd(uint8_t bin)`

Convert a binary value to binary coded decimal (BCD 8421).

Parameters

- `bin` – Binary value to convert.

Returns BCD 8421 representation of input value.

uint8_t `u8_to_dec(char *buf, uint8_t buflen, uint8_t value)`

Convert a uint8_t into a decimal string representation.

Convert a uint8_t value into its ASCII decimal string representation. The string is terminated if there is enough space in buf.

Parameters

- `buf` – Address of where to store the string representation.
- `buflen` – Size of the storage area for string representation.
- `value` – The value to convert to decimal string

Returns The length of the converted string (excluding terminator if any), or 0 if an error occurred.

7.32 Settings

The settings subsystem gives modules a way to store persistent per-device configuration and runtime state. A variety of storage implementations are provided behind a common API using FCB, NVS, or a file system. These different implementations give the application developer flexibility to select an appropriate storage medium, and even change it later as needs change. This subsystem is used by various Zephyr components and can be used simultaneously by user applications.

Settings items are stored as key-value pair strings. By convention, the keys can be organized by the package and subtree defining the key, for example the key `id/serial` would define the `serial` configuration element for the package `id`.

Convenience routines are provided for converting a key value to and from a string type.

For an example of the settings subsystem refer to the sample.

Note: As of Zephyr release 2.1 the recommended backend for non-filesystem storage is *NVS*.

7.32.1 Handlers

Settings handlers for subtree implement a set of handler functions. These are registered using a call to `settings_register()`.

h_get This gets called when asking for a settings element value by its name using `settings_runtime_get()` from the runtime backend.

h_set This gets called when the value is loaded from persisted storage with `settings_load()`, or when using `settings_runtime_set()` from the runtime backend.

h_commit This gets called after the settings have been loaded in full. Sometimes you don't want an individual setting value to take effect right away, for example if there are multiple settings which are interdependent.

h_export This gets called to write all current settings. This happens when `settings_save()` tries to save the settings or transfer to any user-implemented back-end.

7.32.2 Backends

Backends are meant to load and save data to/from setting handlers, and implement a set of handler functions. These are registered using a call to `settings_src_register()` for backends that can load data, and/or `settings_dst_register()` for backends that can save data. The current implementation allows for multiple source backends but only a single destination backend.

csi_load This gets called when loading values from persistent storage using `settings_load()`.

csi_save This gets called when a saving a single setting to persistent storage using `settings_save_one()`.

csi_save_start This gets called when starting a save of all current settings using `settings_save()`.

csi_save_end This gets called after having saved of all current settings using `settings_save()`.

7.32.3 Zephyr Storage Backends

Zephyr has three storage backends: a Flash Circular Buffer (`CONFIG_SETTINGS_FCB`), a file in the filesystem (`CONFIG_SETTINGS_FS`), or non-volatile storage (`CONFIG_SETTINGS_NVS`).

You can declare multiple sources for settings; settings from all of these are restored when `settings_load()` is called.

There can be only one target for writing settings; this is where data is stored when you call `settings_save()`, or `settings_save_one()`.

FCB read target is registered using `settings_fcb_src()`, and write target using `settings_fcb_dst()`. As a side-effect, `settings_fcb_src()` initializes the FCB area, so it must be called before calling `settings_fcb_dst()`. File read target is registered using `settings_file_src()`, and write target by using `settings_file_dst()`. Non-volatile storage read target is registered using `settings_nvs_src()`, and write target by using `settings_nvs_dst()`.

7.32.4 Loading data from persisted storage

A call to `settings_load()` uses an `h_set` implementation to load settings data from storage to volatile memory. After all data is loaded, the `h_commit` handler is issued, signalling the application that the settings were successfully retrieved.

Technically FCB and filesystem backends may store some history of the entities. This means that the newest data entity is stored after any older existing data entities. Starting with Zephyr 2.1, the back-end must filter out all old entities and call the callback with only the newest entity.

7.32.5 Storing data to persistent storage

A call to `settings_save_one()` uses a backend implementation to store settings data to the storage medium. A call to `settings_save()` uses an `h_export` implementation to store different data in one operation using `settings_save_one()`. A key need to be covered by a `h_export` only if it is supposed to be stored by `settings_save()` call.

For both FCB and filesystem back-end only storage requests with data which changes most actual key's value are stored, therefore there is no need to check whether a value changed by the application. Such a storage mechanism implies that storage can contain multiple value assignments for a key , while only the last is the current value for the key.

Garbage collection

When storage becomes full (FCB) or consumes too much space (file system), the backend removes non-recent key-value pairs records and unnecessary key-delete records.

7.32.6 Example: Device Configuration

This is a simple example, where the settings handler only implements `h_set` and `h_export`. `h_set` is called when the value is restored from storage (or when set initially), and `h_export` is used to write the value to storage thanks to `storage_func()`. The user can also implement some other export functionality, for example, writing to the shell console).

```
#define DEFAULT_FOO_VAL_VALUE 1

static int8 foo_val = DEFAULT_FOO_VAL_VALUE;

static int foo_settings_set(const char *name, size_t len,
                            settings_read_cb read_cb, void *cb_arg)
{
    const char *next;
    int rc;

    if (settings_name_steq(name, "bar", &next) && !next) {
        if (len != sizeof(foo_val)) {
            return -EINVAL;
        }

        rc = read_cb(cb_arg, &foo_val, sizeof(foo_val));
        if (rc >= 0) {
            /* key-value pair was properly read.
             * rc contains value length.
             */
            return 0;
        }
        /* read-out error */
        return rc;
    }

    return -ENOENT;
}

static int foo_settings_export(int (*storage_func)(const char *name,
                                                   void *value,
                                                   size_t val_len))
```

(continues on next page)

(continued from previous page)

```
{
    return storage_func("foo/bar", &foo_val, sizeof(foo_val));
}

struct settings_handler my_conf = {
    .name = "foo",
    .h_set = foo_settings_set,
    .h_export = foo_settings_export
};
```

7.32.7 Example: Persist Runtime State

This is a simple example showing how to persist runtime state. In this example, only `h_set` is defined, which is used when restoring value from persisted storage.

In this example, the `main` function increments `foo_val`, and then persists the latest number. When the system restarts, the application calls `settings_load()` while initializing, and `foo_val` will continue counting up from where it was before restart.

```
#include <zephyr.h>
#include <sys/reboot.h>
#include <settings/settings.h>
#include <sys/printk.h>
#include <inttypes.h>

#define DEFAULT_FOO_VAL_VALUE 0

static uint8_t foo_val = DEFAULT_FOO_VAL_VALUE;

static int foo_settings_set(const char *name, size_t len,
                            settings_read_cb read_cb, void *cb_arg)
{
    const char *next;
    int rc;

    if (settings_name_steq(name, "bar", &next) && !next) {
        if (len != sizeof(foo_val)) {
            return -EINVAL;
        }

        rc = read_cb(cb_arg, &foo_val, sizeof(foo_val));
        if (rc >= 0) {
            return 0;
        }

        return rc;
    }


    return -ENOENT;
}

struct settings_handler my_conf = {
    .name = "foo",
    .h_set = foo_settings_set
```

(continues on next page)

(continued from previous page)

```
};

void main(void)
{
    settings_subsys_init();
    settings_register(&my_conf);
    settings_load();

    foo_val++;
    settings_save_one("foo/bar", &foo_val, sizeof(foo_val));

    printk("foo: %d\n", foo_val);

    k_sleep(1000);
    sys_reboot(SYS_REBOOT_COLD);
}
```

7.32.8 Example: Custom Backend Implementation

This is a simple example showing how to register a simple custom backend handler (`CONFIG_SETTINGS_CUSTOM`).

```
static int settings_custom_load(struct settings_store *cs)
{
    //...
}

static int settings_custom_save(struct settings_store *cs, const char *name,
                                const char *value, size_t val_len)
{
    //...
}

/* custom backend interface */
static struct settings_store_itf settings_custom_itf = {
    .csi_load = settings_custom_load,
    .csi_save = settings_custom_save,
};

/* custom backend node */
static struct settings_store settings_custom_store = {
    .cs_itf = &settings_custom_itf
}

int settings_backend_init(void)
{
    /* register custom backend */
    settings_dst_register(&settings_custom_store);
    settings_src_register(&settings_custom_store);
    return 0;
}
```

7.32.9 API Reference

The Settings subsystem APIs are provided by `settings.h`:

API for general settings usage

group `settings`

Defines

`SETTINGS_MAX_DIR_DEPTH`

`SETTINGS_MAX_NAME_LEN`

`SETTINGS_MAX_VAL_LEN`

`SETTINGS_NAME_SEPARATOR`

`SETTINGS_NAME_END`

`SETTINGS_EXTRA_LEN`

`SETTINGS_STATIC_HANDLER_DEFINE(_hname, _tree, _get, _set, _commit, _export)`

Define a static handler for settings items

This creates a variable *hname prepended by* settings_handler.

Parameters

- `_hname` – handler name
- `_tree` – subtree name
- `_get` – get routine (can be NULL)
- `_set` – set routine (can be NULL)
- `_commit` – commit routine (can be NULL)
- `_export` – export routine (can be NULL)

Typedefs

typedef ssize_t (*`settings_read_cb`)(void *cb_arg, void *data, size_t len)

Function used to read the data from the settings storage in h_set handler implementations.

Param cb_arg [in] arguments for the read function. Appropriate cb_arg is transferred to h_set handler implementation by the backend.

Param data [out] the destination buffer

Param len [in] length of read

Return positive: Number of bytes read, 0: key-value pair is deleted. On error returns -ERRNO code.

typedef int (*`settings_load_direct_cb`)(const char *key, size_t len, *settings_read_cb* read_cb, void *cb_arg, void *param)

Callback function used for direct loading. Used by *settings_load_subtree_direct* function.

- key[in] the name with skipped part that was used as name in handler registration
- len[in] the size of the data found in the backend.
- read_cb[in] function provided to read the data from the backend.
- cb_arg[in] arguments for the read function provided by the backend.

Param key **[in]** the name with skipped part that was used as name in handler registration

Param len **[in]** the size of the data found in the backend.

Param read_cb **[in]** function provided to read the data from the backend.

Param cb_arg **[inout]** arguments for the read function provided by the backend.

Param param **[inout]** parameter given to the *settings_load_subtree_direct* function.

Return When nonzero value is returned, further subtree searching is stopped. Use with care as some settings backends would iterate through old values, and the current value is returned last.

Functions

int `settings_subsys_init`(void)

Initialization of settings and backend

Can be called at application startup. In case the backend is a FS Remember to call it after the FS was mounted. For FCB backend it can be called without such a restriction.

Returns 0 on success, non-zero on failure.

int `settings_register`(struct *settings_handler* *cf)

Register a handler for settings items stored in RAM.

Parameters

- `cf` – Structure containing registration info.

Returns 0 on success, non-zero on failure.

int `settings_load`(void)

Load serialized items from registered persistence sources. Handlers for serialized item subtrees registered earlier will be called for encountered values.

Returns 0 on success, non-zero on failure.

int `settings_load_subtree`(const char *subtree)

Load limited set of serialized items from registered persistence sources. Handlers for serialized item subtrees registered earlier will be called for encountered values that belong to the subtree.

Parameters

- `subtree` – **[in]** name of the subtree to be loaded.

Returns 0 on success, non-zero on failure.

int `settings_load_subtree_direct`(const char *subtree, *settings_load_direct_cb* cb, void *param)

Load limited set of serialized items using given callback.

This function bypasses the normal data workflow in settings module. All the settings values that are found are passed to the given callback.

Note: This function does not call commit function. It works as a blocking function, so it is up to the user to call any kind of commit function when this operation ends.

Parameters

- `subtree` – **[in]** subtree name of the subtree to be loaded.
- `cb` – **[in]** pointer to the callback function.
- `param` – **[inout]** parameter to be passed when callback function is called.

Returns 0 on success, non-zero on failure.

int `settings_save`(void)

Save currently running serialized items. All serialized items which are different from currently persisted values will be saved.

Returns 0 on success, non-zero on failure.

int `settings_save_one`(const char *name, const void *value, size_t val_len)

Write a single serialized value to persisted storage (if it has changed value).

Parameters

- `name` – Name/key of the settings item.
- `value` – Pointer to the value of the settings item. This value will be transferred to the *settings_handler::h_export* handler implementation.
- `val_len` – Length of the value.

Returns 0 on success, non-zero on failure.

int `settings_delete`(const char *name)

Delete a single serialized in persisted storage.

Deleting an existing key-value pair in the settings mean to set its value to NULL.

Parameters

- `name` – Name/key of the settings item.

Returns 0 on success, non-zero on failure.

int `settings_commit`(void)

Call commit for all settings handler. This should apply all settings which has been set, but not applied yet.

Returns 0 on success, non-zero on failure.

int `settings_commit_subtree`(const char *subtree)

Call commit for settings handler that belong to subtree. This should apply all settings which has been set, but not applied yet.

Parameters

- `subtree` – **[in]** name of the subtree to be committed.

Returns 0 on success, non-zero on failure.

struct `settings_handler`

#include <settings.h> Config handlers for subtree implement a set of handler functions. These are registered using a call to *settings_register*.

Public Members

const char *`name`

Name of subtree.

int (*`h_get`)(const char *key, char *val, int val_len_max)

Get values handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- val[out] buffer to receive value.
- val_len_max[in] size of that buffer.

Return: length of data read on success, negative on failure.

int (*`h_set`)(const char *key, size_t len, *settings_read_cb* read_cb, void *cb_arg)

Set value handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- len[in] the size of the data found in the backend.
- read_cb[in] function provided to read the data from the backend.
- cb_arg[in] arguments for the read function provided by the backend.

Return: 0 on success, non-zero on failure.

int (*`h_commit`)(void)

This handler gets called after settings has been loaded in full. User might use it to apply setting to the application.

Return: 0 on success, non-zero on failure.

int (*`h_export`)(int (*export_func)(const char *name, const void *val, size_t val_len))

This gets called to dump all current settings items.

This happens when *settings_save* tries to save the settings. Parameters:
- export_func: the pointer to the internal function which appends a single key-value pair to persisted settings. Don't store duplicated value. The name is subtree/key string, val is the string with value.

Return: 0 on success, non-zero on failure.

Remark

The User might limit a implementations of handler to serving only one keyword at one call - what will impose limit to get/set values using full subtree/key name.

---

sys_snode_t `node`

Linked list node info for module internal usage.

struct `settings_handler_static`

#include <settings.h> Config handlers without the node element, used for static handlers. These are registered using a call to SETTINGS_REGISTER_STATIC().

Public Members

const char *`name`

Name of subtree.

int (*`h_get`)(const char *key, char *val, int val_len_max)

Get values handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- val[out] buffer to receive value.
- val_len_max[in] size of that buffer.

Return: length of data read on success, negative on failure.

int (*`h_set`)(const char *key, size_t len, *settings_read_cb* read_cb, void *cb_arg)

Set value handler of settings items identified by keyword names.

Parameters:
- key[in] the name with skipped part that was used as name in handler registration
- len[in] the size of the data found in the backend.
- read_cb[in] function provided to read the data from the backend.
- cb_arg[in] arguments for the read function provided by the backend.

Return: 0 on success, non-zero on failure.

int (*`h_commit`)(void)

This handler gets called after settings has been loaded in full. User might use it to apply setting to the application.

int (*`h_export`)(int (*export_func)(const char *name, const void *val, size_t val_len))

This gets called to dump all current settings items.

This happens when *settings_save* tries to save the settings. Parameters:
- export_func: the pointer to the internal function which appends a single key-value pair to persisted settings. Don't store duplicated value. The name is subtree/key string, val is the string with value.

Return: 0 on success, non-zero on failure.

Remark

The User might limit a implementations of handler to serving only one keyword at one call - what will impose limit to get/set values using full subtree/key name.

API for key-name processing

group `settings_name_proc`

API for const name processing.

Functions

int `settings_name_steq`(const char *name, const char *key, const char **next)

Compares the start of name with a key

Some examples: settings_name_steq("bt/btmesh/iv", "b", &next) returns 1, next="t/btmesh/iv" settings_name_steq("bt/btmesh/iv", "bt", &next) returns 1, next="btmesh/iv" settings_name_steq("bt/btmesh/iv", "bt/", &next) returns 0, next=NULL settings_name_steq("bt/btmesh/iv", "bta", &next) returns 0, next=NULL

REMARK: This routine could be simplified if the *settings_handler* names would include a separator at the end.

Parameters

- `name` – **[in]** in string format
- `key` – **[in]** comparison string
- `next` – **[out]** pointer to remaining of name, when the remaining part starts with a separator the separator is removed from next

Returns 0: no match 1: match, next can be used to check if match is full

int `settings_name_next`(const char *name, const char **next)

determine the number of characters before the first separator

Parameters

- `name` – **[in]** in string format
- `next` – **[out]** pointer to remaining of name (excluding separator)

Returns index of the first separator, in case no separator was found this is the size of name

API for runtime settings manipulation

group `settings_rt`

API for runtime settings.

Functions

int `settings_runtime_set`(const char *name, const void *data, size_t len)

Set a value with a specific key to a module handler.

Parameters

- `name` – Key in string format.
- `data` – Binary value.
- `len` – Value length in bytes.

Returns 0 on success, non-zero on failure.

int `settings_runtime_get`(const char *name, void *data, size_t len)

Get a value corresponding to a key from a module handler.

Parameters

- `name` – Key in string format.

- `data` – Returned binary value.
- `len` – requested value length in bytes.

Returns length of data read on success, negative on failure.

int `settings_runtime_commit`(const char *name)

Apply settings in a module handler.

Parameters

- `name` – Key in string format.

Returns 0 on success, non-zero on failure.

API of backend interface

group `settings_backend`

settings

Functions

void `settings_src_register`(struct *settings_store* *cs)

Register a backend handler acting as source.

Parameters

- `cs` – Backend handler node containing handler information.

void `settings_dst_register`(struct *settings_store* *cs)

Register a backend handler acting as destination.

Parameters

- `cs` – Backend handler node containing handler information.

struct settings_handler_ *`settings_parse_and_lookup`(const char *name, const char **next)

Parses a key to an array of elements and locate corresponding module handler.

Parameters

- `name` – **[in]** in string format
- `next` – **[out]** remaining of name after matched handler

Returns *settings_handler_static* on success, NULL on failure.

int `settings_call_set_handler`(const char *name, size_t len, *settings_read_cb* read_cb, void *read_cb_arg, const struct *settings_load_arg* *load_arg)

Calls settings handler.

Parameters

- `name` – **[in]** The name of the data found in the backend.
- `len` – **[in]** The size of the data found in the backend.
- `read_cb` – **[in]** Function provided to read the data from the backend.
- `read_cb_arg` – **[inout]** Arguments for the read function provided by the backend.
- `load_arg` – **[inout]** Arguments for data loading.

Returns 0 or negative error code

struct `settings_store`

#include <settings.h> Backend handler node for storage handling.

Public Members

sys_snode_t `cs_next`

Linked list node info for internal usage.

const struct *settings_store_itf* *`cs_itf`

Backend handler structure.

struct `settings_load_arg`

#include <settings.h> Arguments for data loading. Holds all parameters that changes the way data should be loaded from backend.

Public Members

const char *`subtree`

Name of the subtree to be loaded.

If NULL, all values would be loaded.

settings_load_direct_cb `cb`

Pointer to the callback function.

If NULL then matching registered function would be used.

void *`param`

Parameter for callback function.

Parameter to be passed to the callback function.

struct `settings_store_itf`

#include <settings.h> Backend handler functions. Sources are registered using a call to *settings_src_register*. Destinations are registered using a call to *settings_dst_register*.

Public Members

int (*`csi_load`)(struct *settings_store* *cs, const struct *settings_load_arg* *arg)

Loads values from storage limited to subtree defined by subtree.

Parameters:
- cs - Corresponding backend handler node,
- arg - Structure that holds additional data for data loading.

Note: Backend is expected not to provide duplicates of the entities. It means that if the backend does not contain any functionality to really delete old keys, it has to filter out old entities and call load callback only on the final entity.

---

int (*csi_save_start)(struct *settings_store* *cs)

Handler called before an export operation.

Parameters:
- cs - Corresponding backend handler node

int (*csi_save)(struct *settings_store* *cs, const char *name, const char *value, size_t val_len)

Save a single key-value pair to storage.

Parameters:
- cs - Corresponding backend handler node
- name - Key in string format
- value - Binary value
- val_len - Length of value in bytes.

int (*csi_save_end)(struct *settings_store* *cs)

Handler called after an export operation.

Parameters:
- cs - Corresponding backend handler node

7.33 Executing Time Functions

The timing functions can be used to obtain execution time of a section of code to aid in analysis and optimization.

Please note that the timing functions may use a different timer than the default kernel timer, where the timer being used is specified by architecture, SoC or board configuration.

7.33.1 Configuration

To allow using the timing functions, CONFIG_TIMING_FUNCTIONS needs to be enabled.

7.33.2 Usage

To gather timing information:

1. Call *timing_init()* to initialize the timer.
2. Call *timing_start()* to signal the start of gathering of timing information. This usually starts the timer.
3. Call *timing_counter_get()* to mark the start of code execution.
4. Call *timing_counter_get()* to mark the end of code execution.
5. Call *timing_cycles_get()* to get the number of timer cycles between start and end of code execution.
6. Call *timing_cycles_to_ns()* with total number of cycles to convert number of cycles to nanoseconds.
7. Repeat from step 3 to gather timing information for other blocks of code.
8. Call *timing_stop()* to signal the end of gathering of timing information. This usually stops the timer.

Example

This shows an example on how to use the timing functions:

```
#include <timing/timing.h>

void gather_timing(void)
{
    timing_t start_time, end_time;
    uint64_t total_cycles;
    uint64_t total_ns;

    timing_init();
    timing_start();

    start_time = timing_counter_get();

    code_execution_to_be_measured();

    end_time = timing_counter_get();

    total_cycles = timing_cycles_get(&start_time, &end_time);
    total_ns = timing_cycles_to_ns(total_cycles);

    timing_stop();
}
```

7.33.3 API documentation

group `timing_api`

Timing Measurement APIs.

Functions

void `timing_init`(void)

Initialize the timing subsystem.

Perform the necessary steps to initialize the timing subsystem.

void `timing_start`(void)

Signal the start of the timing information gathering.

Signal to the timing subsystem that timing information will be gathered from this point forward.

void `timing_stop`(void)

Signal the end of the timing information gathering.

Signal to the timing subsystem that timing information is no longer being gathered from this point forward.

static inline timing_t `timing_counter_get`(void)

Return timing counter.

Returns Timing counter.

static inline uint64_t timing_cycles_get(volatile timing_t *const start, volatile timing_t *const end)

Get number of cycles between `start` and `end`.

For some architectures or SoCs, the raw numbers from counter need to be scaled to obtain actual number of cycles.

Parameters

- `start` – Pointer to counter at start of a measured execution.
- `end` – Pointer to counter at stop of a measured execution.

Returns Number of cycles between start and end.

static inline uint64_t timing_freq_get(void)

Get frequency of counter used (in Hz).

Returns Frequency of counter used for timing in Hz.

static inline uint64_t timing_cycles_to_ns(uint64_t cycles)

Convert number of `cycles` into nanoseconds.

Parameters

- `cycles` – Number of cycles

Returns Converted time value

static inline uint64_t timing_cycles_to_ns_avg(uint64_t cycles, uint32_t count)

Convert number of `cycles` into nanoseconds with averaging.

Parameters

- `cycles` – Number of cycles
- `count` – Times of accumulated cycles to average over

Returns Converted time value

static inline uint32_t timing_freq_get_mhz(void)

Get frequency of counter used (in MHz).

Returns Frequency of counter used for timing in MHz.

7.34 Virtualization

7.34.1 Inter-VM Shared Memory

Overview

As Zephyr is enabled to run as a guest OS on Qemu and ACRN it might be necessary to make VMs aware of each other, or aware of the host. This is made possible by exposing a shared memory among parties via a feature called ivshmem, which stands for inter-VM Shared Memory.

The Two types are supported: a plain shared memory (ivshmem-plain) or a shared memory with the ability for a VM to generate an interruption on another, and thus to be interrupted as well itself (ivshmem-doorbell).

Please refer to the official Qemu ivshmem documentation for more information.

Support

Zephyr supports both version: plain and doorbell. Ivshmem driver can be build by enabling `CONFIG_IVSHMEM`. By default, this will expose the plain version. `CONFIG_IVSHMEM_DOORBELL` needs to be enabled to get the doorbell version.

Because the doorbell version uses MSI-X vectors to support notification vectors, the `CONFIG_IVSHMEM_MSI_X_VECTORS` has to be tweaked to the amount of vectors that will be needed.

Note that a tiny shell module can be exposed to test the ivshmem feature by enabling `CONFIG_IVSHMEM_SHELL`.

API Reference

group `ivshmem`

ivshmem reference API

Typedefs

typedef size_t (*`ivshmem_get_mem_f`)(const struct *device* *dev, uintptr_t *memmap)

typedef uint32_t (*`ivshmem_get_id_f`)(const struct *device* *dev)

typedef uint16_t (*`ivshmem_get_vectors_f`)(const struct *device* *dev)

typedef int (*`ivshmem_int_peer_f`)(const struct *device* *dev, uint32_t peer_id, uint16_t vector)

typedef int (*`ivshmem_register_handler_f`)(const struct *device* *dev, struct *k_poll_signal* *signal, uint16_t vector)

Functions

size_t `ivshmem_get_mem`(const struct *device* *dev, uintptr_t *memmap)

Get the inter-VM shared memory.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `memmap` – A pointer to fill in with the memory address

Returns the size of the memory mapped, or 0

uint32_t `ivshmem_get_id`(const struct *device* *dev)

Get our VM ID.

Parameters

- `dev` – Pointer to the device structure for the driver instance

Returns our VM ID or 0 if we are not running on doorbell version

uint16_t `ivshmem_get_vectors`(const struct *device* *dev)

Get the number of interrupt vectors we can use.

Parameters

- `dev` – Pointer to the device structure for the driver instance

Returns the number of available interrupt vectors

int `ivshmem_int_peer`(const struct *device* *dev, uint32_t peer_id, uint16_t vector)

Interrupt another VM.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `peer_id` – The VM ID to interrupt
- `vector` – The interrupt vector to use

Returns 0 on success, a negative errno otherwise

int `ivshmem_register_handler`(const struct *device* *dev, struct *k_poll_signal* *signal, uint16_t vector)

Register a vector notification (interrupt) handler.

Note: The returned status, if positive, to a raised signal is the vector that generated the signal. This lets the possibility to the user to have one signal for all vectors, or one per-vector.

Parameters

- `dev` – Pointer to the device structure for the driver instance
- `signal` – A pointer to a valid and ready to be signaled struct *k_poll_signal*. Or NULL to unregister any handler registered for the given vector.
- `vector` – The interrupt vector to get notification from

Returns 0 on success, a negative errno otherwise

struct `ivshmem_driver_api`

#include <ivshmem.h>

Chapter 8

User and Developer Guides

8.1 Beyond the Getting Started Guide

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

8.1.1 Python and pip

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

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

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

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

8.1.2 Advanced Setup and tool chain alternatives

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

[1] pip is Python's package installer. Its `install` command first tries to re-use packages and package dependencies already installed on your computer. If that is not possible, `pip install` downloads them from the Python Package Index (PyPI) on the Internet.

The package versions requested by Zephyr's `requirements.txt` may conflict with other requirements on your system, in which case you may want to set up a virtualenv for Zephyr development.

Install Linux Host Dependencies

Documentation is available for these Linux distributions:

- Ubuntu
- Fedora
- Clear Linux
- Arch Linux

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

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

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

Ubuntu

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

Fedora

```
sudo dnf upgrade
```

Clear Linux

```
sudo swupd update
```

Arch Linux

```
sudo pacman -Syu
```

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

Ubuntu

```
sudo apt-get install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils file␣
↪libpython3.8-dev \
  make gcc gcc-multilib g++-multilib libsdl2-dev
```

Fedora

```
sudo dnf group install "Development Tools" "C Development Tools and Libraries"
dnf install git cmake ninja-build gperf ccache dfu-util dtc wget \
  python3-pip python3-tkinter xz file glibc-devel.i686 libstdc++-devel.i686 python38 \
  SDL2-devel
```

Clear Linux

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

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

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

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

Arch Linux

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

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

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

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

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

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

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

- Use `pip3`:

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

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

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

```
sudo snap install cmake
```

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

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

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

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

Install the Zephyr Software Development Kit (SDK) Use of the Zephyr SDK is optional, but recommended. Some of the dependencies installed above are only needed for installing the SDK.

Zephyr's SDK (Software Development Kit) contains all necessary tools to build Zephyr on all supported architectures. Additionally, it includes host tools such as custom QEMU binaries and a host compiler. The SDK supports the following target architectures:

- X86 (Intel Architecture 32 bits)
- ARM (Advanced RISC Machine)
- ARC (Argonaut RISC Core)
- NIOS II
- RISC-V
- SPARC
- XTENSA

Follow these steps to install the Zephyr SDK:

1. Download the latest SDK as a self-extracting installation binary:

   ```
   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.13.1/
   ↪zephyr-sdk-0.13.1-linux-x86_64-setup.run
   ```

   (You can change *0.13.1* to another version if needed; the Zephyr Downloads page contains all available SDK releases.)

2. Run the installation binary, installing the SDK at `~/zephyr-sdk-0.13.1`:

   ```
   cd <sdk download directory>
   chmod +x zephyr-sdk-0.13.1-linux-x86_64-setup.run
   ./zephyr-sdk-0.13.1-linux-x86_64-setup.run -- -d ~/zephyr-sdk-0.13.1
   ```

   You can pick another directory if you want. If this fails, make sure Zephyr's dependencies were installed as described in *Install Requirements and Dependencies*.

If you ever want to uninstall the SDK, just remove the directory where you installed it.

Note: It is recommended to install the Zephyr SDK at one of the following locations:

- `$HOME/zephyr-sdk[-x.y.z]`

- `$HOME/.local/zephyr-sdk[-x.y.z]`
- `$HOME/.local/opt/zephyr-sdk[-x.y.z]`
- `$HOME/bin/zephyr-sdk[-x.y.z]`
- `/opt/zephyr-sdk[-x.y.z]`
- `/usr/zephyr-sdk[-x.y.z]`
- `/usr/local/zephyr-sdk[-x.y.z]`

where `[-x.y.z]` is optional text, and can be any text, for example `-0.13.1`.

If you install the Zephyr SDK outside any of those locations, then it is required to register the Zephyr SDK in the CMake package registry during installation or set `ZEPHYR_SDK_INSTALL_DIR` to point to the Zephyr SDK installation folder.

`ZEPHYR_SDK_INSTALL_DIR` can also be used for pointing to a folder containing multiple Zephyr SDKs, allowing for automatic toolchain selection, for example: `ZEPHYR_SDK_INSTALL_DIR=/company/tools`

- `/company/tools/zephyr-sdk-0.13.1`
- `/company/tools/zephyr-sdk-a.b.c`
- `/company/tools/zephyr-sdk-x.y.z`

this allow Zephyr to pick the right toolchain, while allowing multiple Zephyr SDKs to be grouped together at a custom location.

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

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

To make sure this variable is unset, run:

```
unset ZEPHYR_SDK_INSTALL_DIR
```

macOS alternative setup instructions

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

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

Note that this section does **not** apply to executables installed with Homebrew, since those are automatically un-quarantined by `brew` itself. This is however relevant for most *3rd Party Toolchains*.

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

Windows alternative setup instructions

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

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

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

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

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

8.1.3 Set Up a Toolchain

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

On Linux systems, you can install the *Zephyr SDK* to get toolchains for all supported architectures. Otherwise, you can install other toolchains in the usual way for your operating system: with installer programs or system package managers, by downloading and extracting a zip archive, etc.

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

While the Zephyr SDK includes standard tool chains for all supported architectures, there are also customized alternatives as described in these documents. (If you're not sure which to use, check your specific board-level documentation. If you're targeting an Arm Cortex-M board, for example, *GNU Arm Embedded* is a safe bet.)

3rd Party Toolchains

A "3rd party toolchain" is an officially supported toolchain provided by an external organization. Several of these are available.

GNU Arm Embedded

Warning: Do not install the toolchain into a path with spaces.

1. Download and install a GNU Arm Embedded build for your operating system and extract it on your file system.

   Note: On Windows, we'll assume you install into the directory `C:\gnu_arm_embedded`.

   Warning: On macOS Catalina or later you might need to *change a security policy* for the toolchain to be able to run from the terminal.

2. *Set these environment variables*:
   - Set `ZEPHYR_TOOLCHAIN_VARIANT` to `gnuarmemb`.
   - Set `GNUARMEMB_TOOLCHAIN_PATH` to the toolchain installation directory.
3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `GNUARMEMB_TOOLCHAIN_PATH` values may be different on your system):

```
# Linux, macOS:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
gnuarmemb
$ echo $GNUARMEMB_TOOLCHAIN_PATH
/home/you/Downloads/gnu_arm_embedded

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
gnuarmemb
> echo %GNUARMEMB_TOOLCHAIN_PATH%
C:\gnu_arm_embedded
```

   Warning: On macOS, if you are having trouble with the suggested procedure, there is an unofficial package on brew that might help you. Run `brew install gcc-arm-embedded` and configure the variables
   - Set `ZEPHYR_TOOLCHAIN_VARIANT` to `gnuarmemb`.
   - Set `GNUARMEMB_TOOLCHAIN_PATH` to the brew installation directory (something like `/usr/local`)

Arm Compiler 6

1. Download and install a development suite containing the Arm Compiler 6 for your operating system.
2. *Set these environment variables*:
   - Set `ZEPHYR_TOOLCHAIN_VARIANT` to `armclang`.
   - Set `ARMCLANG_TOOLCHAIN_PATH` to the toolchain installation directory.
3. The Arm Compiler 6 needs the `ARMLMD_LICENSE_FILE` environment variable to point to your license file or server.

For example:

```
# Linux, macOS, license file:
export ARMLMD_LICENSE_FILE=/<path>/license_armds.dat
# Linux, macOS, license server:
export ARMLMD_LICENSE_FILE=8224@myserver
```

(continues on next page)

(continued from previous page)

```
# Windows, license file:
> set ARMLMD_LICENSE_FILE=c:\<path>\license_armds.dat
# Windows, license server:
> set ARMLMD_LICENSE_FILE=8224@myserver
```

1. If the Arm Compiler 6 was installed as part of an Arm Development Studio, then you must set the `ARM_PRODUCT_DEF` to point to the product definition file: See also: Product and toolkit configuration. For example if the Arm Development Studio is installed in: `/opt/armds-2020-1` with a Gold license, then set `ARM_PRODUCT_DEF` to point to `/opt/armds-2020-1/gold.elmap`.

Note: The Arm Compiler 6 uses `armlink` for linking. This is incompatible with Zephyr's linker script template, which works with GNU ld. Zephyr's Arm Compiler 6 support Zephyr's CMake linker script generator, which supports generating scatter files. Basic scatter file support is in place, but there are still areas covered in ld templates which are not fully supported by the CMake linker script generator.

Some Zephyr subsystems or modules may also contain C or assembly code that relies on GNU intrinsics and have not yet been updated to work fully with `armclang`.

Intel oneAPI Toolkit

1. Download Intel oneAPI Base Toolkit
2. Assuming the toolkit is installed in `/opt/intel/oneApi`, set environment using:

   ```
   # Linux, macOS:
   export ONEAPI_TOOLCHAIN_PATH=/opt/intel/oneapi
   source $ONEAPI_TOOLCHAIN_PATH/compiler/latest/env/vars.sh

   # Windows:
   > set ONEAPI_TOOLCHAIN_PATH=C:\Users\Intel\oneapi
   ```

   To setup the complete oneApi environment, use:

   ```
   source  /opt/intel/oneapi/setvars.sh
   ```

   The above will also change the python environment to the one used by the toolchain and might conflict with what Zephyr uses.

3. Set `ZEPHYR_TOOLCHAIN_VARIANT` to `oneApi`.

DesignWare ARC MetaWare Development Toolkit (MWDT)

1. You need to have ARC MWDT installed on your host.
2. *Set these environment variables*:
   - Set `ZEPHYR_TOOLCHAIN_VARIANT` to `arcmwdt`.
   - Set `ARCMWDT_TOOLCHAIN_PATH` to the toolchain installation directory. MWDT installation provides `METAWARE_ROOT` so simply set `ARCMWDT_TOOLCHAIN_PATH` to `$METAWARE_ROOT/../` (Linux) or `%METAWARE_ROOT%\..\` (Windows)
3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `ARCMWDT_TOOLCHAIN_PATH` values may be different on your system):

```
# Linux:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
arcmwdt
$ echo $ARCMWDT_TOOLCHAIN_PATH
/home/you/ARC/MWDT_2019.12/

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
arcmwdt
> echo %ARCMWDT_TOOLCHAIN_PATH%
C:\ARC\MWDT_2019.12\
```

Crosstool-NG You can build toolchains from source code using crosstool-NG.

1. Follow the steps on the crosstool-NG website to prepare your host.
2. Follow the Zephyr SDK with Crosstool NG instructions to build your toolchain. Repeat as necessary to build toolchains for multiple target architectures.

   You will need to clone the `sdk-ng` repo and run the following command:

   ```
   ./go.sh <arch>
   ```

   Note: Currently, only i586 and Arm toolchain builds are verified.

3. *Set these environment variables*:
   - Set `ZEPHYR_TOOLCHAIN_VARIANT` to `xtools`.
   - Set `XTOOLS_TOOLCHAIN_PATH` to the toolchain build directory.
4. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `XTOOLS_TOOLCHAIN_PATH` values may be different on your system):

   ```
   # Linux, macOS:
   $ echo $ZEPHYR_TOOLCHAIN_VARIANT
   xtools
   $ echo $XTOOLS_TOOLCHAIN_PATH
   /Volumes/CrossToolNGNew/build/output/
   ```

Other Cross Compilers

This toolchain variant is borrowed from the Linux kernel build system's mechanism of using a `CROSS_COMPILE` environment variable to set up a GNU-based cross toolchain.

Examples of such "other cross compilers" are cross toolchains that your Linux distribution packaged, that you compiled on your own, or that you downloaded from the net. Unlike toolchains specifically listed in *3rd Party Toolchains*, the Zephyr build system may not have been tested with them, and doesn't officially support them. (Nonetheless, the toolchain set-up mechanism itself is supported.)

Follow these steps to use one of these toolchains.

1. Install a cross compiler suitable for your host and target systems.

   For example, you might install the `gcc-arm-none-eabi` package on Debian-based Linux systems, or `arm-none-eabi-newlib` on Fedora or Red Hat:

```
# On Debian or Ubuntu
sudo apt-get install gcc-arm-none-eabi
# On Fedora or Red Hat
sudo dnf install arm-none-eabi-newlib
```

2. *Set these environment variables*:
   - Set ZEPHYR_TOOLCHAIN_VARIANT to cross-compile.
   - Set CROSS_COMPILE to the common path prefix which your toolchain's binaries have, e.g. the path to the directory containing the compiler binaries plus the target triplet and trailing dash.
3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the CROSS_COMPILE value may be different on your system):

```
# Linux, macOS:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
cross-compile
$ echo $CROSS_COMPILE
/usr/bin/arm-none-eabi-
```

   You can also set CROSS_COMPILE as a CMake variable.

When using this option, all of your toolchain binaries must reside in the same directory and have a common file name prefix. The CROSS_COMPILE variable is set to the directory concatenated with the file name prefix. In the Debian example above, the gcc-arm-none-eabi package installs binaries such as arm-none-eabi-gcc and arm-none-eabi-ld in directory /usr/bin/, so the common prefix is /usr/bin/arm-none-eabi- (including the trailing dash, -). If your toolchain is installed in /opt/mytoolchain/bin with binary names based on target triplet myarch-none-elf, CROSS_COMPILE would be set to /opt/mytoolchain/bin/myarch-none-elf-.

Host Toolchains

In some specific configurations, like when building for non-MCU x86 targets on a Linux host, you may be able to re-use the native development tools provided by your operating system.

To use your host gcc, set the ZEPHYR_TOOLCHAIN_VARIANT *environment variable* to host. To use clang, set ZEPHYR_TOOLCHAIN_VARIANT to llvm.

Custom CMake Toolchains

To use a custom toolchain defined in an external CMake file, *set these environment variables*:

- Set ZEPHYR_TOOLCHAIN_VARIANT to your toolchain's name
- Set TOOLCHAIN_ROOT to the path to the directory containing your toolchain's CMake configuration files.

Zephyr will then include the toolchain cmake files located in the TOOLCHAIN_ROOT directory:

- cmake/toolchain/<toolchain name>/generic.cmake: configures the toolchain for "generic" use, which mostly means running the C preprocessor on the generated *Devicetree* file.
- cmake/toolchain/<toolchain name>/target.cmake: configures the toolchain for "target" use, i.e. building Zephyr and your application's source code.

Here <toolchain name> is the same as the name provided in ZEPHYR_TOOLCHAIN_VARIANT See the zephyr files cmake/generic_toolchain.cmake and cmake/target_toolchain.cmake for more details on what your generic.cmake and target.cmake files should contain.

You can also set ZEPHYR_TOOLCHAIN_VARIANT and TOOLCHAIN_ROOT as CMake variables when generating a build system for a Zephyr application, like so:

```
west build ... -- -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=...
```

```
cmake -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=...
```

If you do this, `-C <initial-cache>` cmake option may useful. If you save your `ZEPHYR_TOOLCHAIN_VARIANT`, `TOOLCHAIN_ROOT`, and other settings in a file named `my-toolchain.cmake`, you can then invoke cmake as `cmake -C my-toolchain.cmake ...` to save typing.

Zephyr includes `include/toolchain.h` which again includes a toolchain specific header based on the compiler identifier, such as `__llvm__` or `__GNUC__`. Some custom compilers identify themselves as the compiler on which they are based, for example `llvm` which then gets the `toolchain/llvm.h` included. This included file may though not be right for the custom toolchain. In order to solve this, and thus to get the `include/other.h` included instead, add the set(TOOLCHAIN_USE_CUSTOM 1) cmake line to the generic.cmake and/or target.cmake files located under `<TOOLCHAIN_ROOT>/cmake/toolchain/<toolchain name>/`.

When `TOOLCHAIN_USE_CUSTOM` is set, the `other.h` must be available out-of-tree and it must include the correct header for the custom toolchain. A good location for the `other.h` header file, would be a directory under the directory specified in `TOOLCHAIN_ROOT` as `include/toolchain`. To get the toolchain header included in zephyr's build, the `USERINCLUDE` can be set to point to the include directory, as shown here:

```
west build -- -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=... -DUSERINCLUDE=...
```

8.1.4 Cloning the Zephyr Repositories

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

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

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

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

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

Keeping Zephyr updated

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

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

8.1.5 Export Zephyr CMake package

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

8.1.6 Board Aliases

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

```
# Variable foo_BOARD_ALIAS=bar replaces BOARD=foo with BOARD=bar and
# sets BOARD_ALIAS=foo in the CMake cache.
set(pca10028_BOARD_ALIAS nrf51dk_nrf51422)
set(pca10056_BOARD_ALIAS nrf52840dk_nrf52840)
set(k64f_BOARD_ALIAS frdm_k64f)
set(sltb004a_BOARD_ALIAS efr32mg_sltb004a)
```

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

8.1.7 Build and Run an Application

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

Build Blinky

Let's build the blinky-sample sample application.

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

1. Go to the zephyr repository:

   ```
   cd zephyrproject/zephyr
   ```

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

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

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

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

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

---

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

Run the Application by Flashing to a Board

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

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

Setting udev rules

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

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

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

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

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

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

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

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

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

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

Run the Application in QEMU

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

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

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

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

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

Run a Sample Application natively (POSIX OS)

You can compile some samples to run as host processes on a POSIX OS. This is currently only tested on Linux hosts. See native_posix for more information. On 64-bit host operating systems, you need to install a 32-bit C library; see native_posix_deps for details.

First, build Hello World for `native_posix`.

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

Next, run the application.

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

Press `Ctrl-C` to exit.

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

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

8.2 Architecture-related Guides

8.2.1 Zephyr support status on ARC processors

Overview

This page describes current state of Zephyr for ARC processors and some future plans. Please note that

- plans are given without exact deadlines
- software features require corresponding hardware to be present and configured the proper way
- not all the features can be enabled at the same time

Support status

Legend: **Y** - yes, supported; **N** - no, not supported; **WIP** - Work In Progress; **TBD** - to be decided

	Processor families			
	EM	**HS3x/4x**	**EV**	**HS6x**
Port status	up-streamed	up-streamed	WIP	up-streamed
Features				
Closely coupled memories (ICCM, DCCM)[1]	Y	Y	TBD	TBD
Execution with caches - Instruction/Data, L1/L2 caches	Y	Y	Y	Y
Hardware-assisted unaligned memory access	Y[2]	Y	TBD	Y
Regular interrupts with multiple priority levels, direct interrupts	Y	Y	TBD	Y
Fast interrupts, separate register banks for fast interrupts	Y	Y	TBD	N
Hardware floating point unit (FPU)	Y	Y	N	TBD
Symmetric multiprocessing (SMP) support, switch-based	N/A	Y	TBD	Y
Hardware-assisted stack checking	Y	Y	TBD	N
Hardware-assisted atomic operations	N/A	Y	TBD	Y
DSP ISA	Y	N[3]	TBD	TBD
DSP AGU/XY extensions	N[Page 1471, 3]	N[Page 1471, 3]	TBD	TBD
Userspace	Y	Y	N	TBD
Memory protection unit (MPU)	Y	Y	TBD	N
Memory management unit (MMU)	N/A	N	N/A	N
SecureShield	Y	N/A	N/A	N/A
Toolchains				
GNU (open source GCC-based)	Y	Y	N	Y
MetaWare (proprietary Clang-based)	Y	Y	Y	WIP[4]
Simulators				
QEMU (open source)[5]	Y	Y	N	Y
nSIM (proprietary, provided by MetaWare Development Tools)	Y	Y	Y	Y

Notes

8.2.2 Arm Cortex-M Developer Guide

Overview

This page contains detailed information about the status of the Arm Cortex-M architecture porting in the Zephyr RTOS and describes key aspects when developing Zephyr applications for Arm Cortex-M-based platforms.

Key supported features

The table below summarizes the status of key OS features in the different Arm Cortex-M implementation variants.

[1] usage of CCMs is limited on SMP systems
[2] except the systems with secure features (SecureShield) due to HW limitation
[3] We only support save/restore ACCL/ACCH registers in task's context. Rest of DSP/AGU registers save/restore isn't implemented but kernel itself does not use these registers. This allows single task per core to use DSP/AGU safely.
[4] MetaWare toolchain supports building for ARCv3 HS6x, however, it's not integrated to Zephyr itself
[5] QEMU doesn't support all the ARC processor's HW features. For the detailed info please check the ARC QEMU documentation

		Processor families							
Architecture variant		Arm v6-M		Arm v7-M			Arm v8-M		Arm v8.1-M
		M0/M1	**M0+**	**M3**	**M4**	**M7**	**M23**	**M33**	**M55**
OS Features									
Programmable fault IRQ priorities		Y	N	Y	Y	Y	N	Y	Y
Single-thread kernel support		Y	Y	Y	Y	Y	Y	Y	Y
Thread local storage support		Y	Y	Y	Y	Y	Y	Y	Y
Interrupt handling									
	Regular interrupts	Y	Y	Y	Y	Y	Y	Y	Y
	Dynamic interrupts	Y	Y	Y	Y	Y	Y	Y	Y
	Direct interrupts	Y	Y	Y	Y	Y	Y	Y	Y
	Zero Latency interrupts	N	N	Y	Y	Y	Y	Y	Y
CPU idling		Y	Y	Y	Y	Y	Y	Y	Y
Native system timer (SysTick)		N[1]	Y	Y	Y	Y	Y	Y	Y
Memory protection									
	User mode	N	Y	Y	Y	Y	Y	Y	Y
	HW stack protection (MPU)	N	N	Y	Y	Y	Y	Y	Y
	HW-assisted stack limit checking	N	N	N	N	N	Y[2]	Y	Y
HW-assisted null-pointer dereference detection		N	N	Y	Y	Y	Y	Y	Y
HW-assisted atomic operations		N	N	Y	Y	Y	N	Y	Y
Support for non-cacheable regions		N	N	Y	Y	Y	N	Y	Y
Execute SRAM functions		N	N	Y	Y	Y	N	Y	Y
Floating Point Services		N	N	N	Y	Y	N	Y	Y
DSP ISA		N	N	N	Y	Y	N	Y	Y
Trusted-Execution									
	Native TrustZone-M support	N	N	N	N	N	Y	Y	Y
	TF-M integration	N	N	N	N	N	N	Y	N
Code relocation		Y	Y	Y	Y	Y	Y	Y	Y
SW-based vector table relaying		Y	Y	Y	Y	Y	Y	Y	Y
HW-assisted timing functions		N	N	Y	Y	Y	N	Y	Y

Notes

[1] SysTick is optional in Cortex-M1
[2] Stack limit checking only in Secure builds in Cortex-M23

OS features

Threads

Thread stack alignment Each Zephyr thread is defined with its own stack memory. By default, Cortex-M enforces a double word thread stack alignment, see `CONFIG_STACK_ALIGN_DOUBLE_WORD`. If MPU-based HW-assisted stack overflow detection (`CONFIG_MPU_STACK_GUARD`) is enabled, thread stacks need to be aligned with a larger value, reflected by `CONFIG_ARM_MPU_REGION_MIN_ALIGN_AND_SIZE`. In Arm v6-M and Arm v7-M architecture variants, thread stacks are additionally required to be align with a value equal to their size, in applications that need to support user mode (`CONFIG_USERSPACE`). The thread stack sizes in that case need to be a power of two. This is all reflected by `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT`, that is enforced in Arm v6-M and Arm v7-M builds with user mode support.

Stack pointers While executing in thread mode the processor is using the Process Stack Pointer (PSP). The processor uses the Main Stack Pointer (MSP) while executing in handler mode, that is, while servicing exceptions and HW interrupts. Using PSP in thread mode *facilitates thread stack pointer manipulation* during thread context switching, without affecting the current execution context flow in handler mode.

In Arm Cortex-M builds a single interrupt stack memory is shared among exceptions and interrupts. The size of the interrupt stack needs to be selected taking into consideration nested interrupts, each pushing an additional stack frame. Deverlopers can modify the interrupt stack size using `CONFIG_ISR_STACK_SIZE`.

The interrupt stack is also used during early boot so the kernel can initialize the main thread's stack before switching to the main thread.

Thread context switching In Arm Cortex-M builds, the PendSV exception is used in order to trigger a context switch to a different thread. PendSV exception is always present in Cortex-M implementations. PendSV is configured with the lowest possible interrupt priority level, in all Cortex-M variants. The main reasons for that design are

- to utilize the tail chaining feature of Cortex-M processors, and thus limit the number of context switch operations that occur.
- to not impact the interrupt latency observed by HW interrupts.

As a result, context switch in Cortex-M is non-atomic, i.e. it may be *preempted* by HW interrupts, however, a context-switch operation must be completed before a new thread context-switch may start.

Typically a thread context-switch will perform the following operations

- When switching-out the current thread, the processor stores
  - the callee-saved registers (R4 - R11) in the thread's container for callee-saved registers, which is located in kernel memory
  - the thread's current operation *mode*
    - user or privileged execution mode
    - presense of an active floating point context
    - the EXC_RETURN value of the current handler context (PendSV)
  - the floating point callee-saved registers (S16 - S31) in the thread's container for FP callee-saved registers, if the current thread has an active FP context
  - the PSP of the current thread which points to the beginning of the current thread's exception stack frame. The latter contains the caller-saved context and the return address of the switched-out thread.
- When switching-in a new thread the processor

- restores the new thread's callee-saved registers from the thread's container for callee-saved registers
- restores the new thread's operation *mode*
- restores the FP callee-saved registers if the switched-in thread had an active FP context before being switched-out
- re-programs the dynamic MPU regions to allow a user thread access its stack and application memories, and/or programs a stack-overflow MPU guard at the bottom of the thread's privileged stack
- restores the PSP for the incoming thread and re-programs the stack pointer limit register (if applicable, see `CONFIG_BUILTIN_STACK_GUARD`)
- optionally does a stack limit checking for the switched-in thread, if sentinel-based stack limit checking is enabled (see `CONFIG_STACK_SENTINEL`).

PendSV exception return sequence restores the new thread's caller-saved registers and the return address, as part of unstacking the exception stack frame.

The implementation of the context-switch mechanism is present in `arch/arm/core/aarch32/swap_helper.S`.

Stack limit checking (Arm v8-M) Armv8-M and Armv8.1-M variants support stack limit checking using the MSPLIM and PSPLIM core registers. The feature is enabled when `CONFIG_BUILTIN_STACK_GUARD` is set. When stack limit checking is enabled, both the thread's privileged or user stack, as well as the interrupt stack are guarded by PSPLIM and MSPLIM registers, respectively. MSPLIM is configured *once* during kernel boot, while PSLIM is re-programmed during every thread context-switch or during system calls, when the thread switches from using its default stack to using its privileged stack, and vice versa. PSPLIM re-programming

- has a relatively low runtime overhead (programming is done with MSR instructions)
- does not impact interrupt latency
- does not require any memory areas to be reserved for stack guards
- does not make use of MPU regions

It is, therefore, considered as a lightweight but very efficient stack overflow detection mechanism in Cortex-M applications.

Stack overflows trigger the dedicated UsageFault exception provided by Arm v8-M.

Interrupt handling features This section describes certain aspects around exception and interrupt handling in Arm Cortex-M.

Interrupt priority levels The number of available (configurable) interrupt priority levels is determined by the number of implemented interrupt priority bits in NVIC; this needs to be described for each Cortex-M platform using DeviceTree:

```
&nvic {
        arm,num-irq-priority-bits = <#priority-bits>;
};
```

Reserved priority levels A number of interrupt priority levels are reserved for the OS.

By design, system fault exceptions have the highest priority level. In *Baseline* Cortex-M, this is actually enforced by hardware, as HardFault is the only available processor fault exception, and its priority is higher than any configurable exception priority.

In *Mainline* Cortex-M, the available fault exceptions (e.g. MemManageFault, UsageFault, etc.) are assigned the highest *configurable* priority level. (`CONFIG_CPU_CORTEX_M_HAS_PROGRAMMABLE_FAULT_PRIOS` signifies explicitly that the Cortex-M implementation supports configurable fault priorities.)

This priority level is never shared with HW interrupts (an exception to this rule is described below). As a result, processor faults occurring in regular ISRs will be handled by the corresponding fault handler and will not escalate to a HardFault, *similar to processor faults occurring in thread mode*.

SVC exception is normally configured with the highest conigurable priority level (an exception to this rule will be described below). SVCs are used by the Zephyr kernel to dispatch system calls, trigger runtime system errors (e.g. Kernel oops or panic), or implement IRQ offloading.

In Baseline Cortex-M the priority level of SVC may be shared with other exceptions or HW interrupts that are also given the highest configurable priority level (As a result of this, kernel runtime errors during interrupt handling will escalate to HardFault. Additional logic in the fault handling routines ensures that such runtime errors are detected successfully).

In Mainline Cortex-M, however, the SVC priority level is *reserved*, thus normally it is only shared with the fault exceptions of configurable priority. This simplifies the fault handling routines in Mainline Cortex-M architecture, since runtime kernel errors are serviced by the SVC handler (i.e no HardFault escalation, even if the kernel errors occur in ISR context).

HW interrupts in Mainline Cortex-M builds are allocated a priority level lower than the SVC.

One exception to the above rules is when Zephyr applications support Zero Latency Interrupts (ZLIs). Such interrupts are designed to have a priority level higher than any HW or system interrupt. If the ZLI feature is enabled in Mainline Cortex-M builds (see `CONFIG_ZERO_LATENCY_IRQS`), then

- ZLIs are assigned the highest configurable priority level
- SVCs are assigned the second highest configurable priority level
- Regular HW interrupts are assigned priority levels lower than SVC.

The priority level configuration in Cortex-M is implemented in `include/arch/arm/aarch32/exc.h`.

Locking and unlocking IRQs In Baseline Cortex-M locking interrupts is implemented using the PRIMASK register.

```
arch_irq_lock()
```

will set the PRIMASK register to 1, eventually, masking all IRQs with configurable priority. While this fulfils the OS requirement of locking interrupts, the consequence is that kernel runtime errors (triggering SVCs) will escalate to HardFault.

In Mainline Cortex-M locking interrupts is implemented using the BASEPRI register (Mainline Cortex-M builds select `CONFIG_CPU_CORTEX_M_HAS_BASEPRI` to signify that BASEPRI register is implemented.). By modifying BASEPRI (or BASEPRI_MAX) arch_irq_lock() masks all system and HW interrupts with the exception of

- SVCs
- processor faults
- ZLIs

This allows zero latency interrupts to be triggered inside OS critical sections. Additionally, this allows system (processor and kernel) faults to be handled by Zephyr in *exactly the same way*, regardless of whether IRQs have been locked or not when the error occurs. It also allows for system calls to be dispatched while IRQs are locked.

Note: Mainline Cortex-M fault handling is designed and configured in a way that all processor and kernel faults are handled by the corresponding exception handlers and never result in HardFault escalation. In other words, a HardFault may only occur in Zephyr applications that have modified the default

fault handling configurations. The main reason for this design was to reserve the HardFault exception for handling exceptional error conditions in safety critical applications.

Dynamic direct interrupts Cortex-M builds support the installation of direct interrupt service routines during runtime. Direct interrupts are designed for performance-critical interrupt handling and do not go through all of the common Zephyr interrupt handling code.

Direct dynamic interrupts are enabled via switching on `CONFIG_DYNAMIC_DIRECT_INTERRUPTS`.

Note that enabling direct dynamic interrupts requires enabling support for dynamic interrupts in the kernel, as well (see `CONFIG_DYNAMIC_INTERRUPTS`).

Zero Latency interrupts As described above, in Mainline Cortex-M applications, the Zephyr kernel reserves the highest configurable interrupt priority level for its own use (SVC). SVCs will not be masked by interrupt locking. Zero-latency interrupt can be used to set up an interrupt at the highest interrupt priority which will not be blocked by interrupt locking. To use the ZLI feature `CONFIG_ZERO_LATENCY_IRQS` needs to be enabled.

Zero latency IRQs have minimal interrupt latency, as they will always preempt regular HW or system interrupts.

Note, however, that since ZLI ISRs will run at a priority level higher than the kernel exceptions they **cannot use** any kernel functionality. Additionally, since the ZLI interrupt priority level is equal to processor fault priority level, faults occurring in ZLI ISRs will escalate to HardFault and will not be handled in the same way as regular processor faults. Developers need to be aware of this limitation.

CPU Idling The Cortex-M architecture port implements both k_cpu_idle() and k_cpu_atomic_idle(). The implementation is present in `arch/arm/core/aarch32/cpu_idle.S`.

In both implementations, the processor will attempt to put the core to low power mode. In k_cpu_idle() the processor ends up executing WFI (Wait For Interrupt) instruction, while in k_cpu_atomic_idle() the processor will execute a WFE (Wait For Event) instruction.

When using the CPU idling API in Cortex-M it is important to note the following:

- Both k_cpu_idle() and k_cpu_atomic_idle() are *assumed* to be invoked with interrupts locked. This is taken care of by the kernel if the APIs are called by the idle thread.
- After waking up from low power mode, both functions will *restore* interrupts unconditionally, that is, regardless of the interrupt lock status before the CPU idle API was called.

The Zephyr CPU Idling mechanism is detailed in *CPU Idling*.

Memory protection features This section describes certain aspects around memory protection features in Arm Cortex-M applications.

User mode system calls User mode is supported in Cortex-M platforms that implement the standard (Arm) MPU or a similar core peripheral logic for memory access policy configuration and control, such as the NXP MPU for Kinetis platforms. (Currently, `CONFIG_ARCH_HAS_USERSPACE` is selected if `CONFIG_ARM_MPU` is enabled by the user in the board default Kconfig settings).

A thread performs a system call by triggering a (synchronous) SVC exception, where

- up to 5 arguments are placed on registers R1 - R5
- system call ID is placed on register R6.

The SVC Handler will branch to the system call preparation logic, which will perform the following operations

- switch the thread's PSP to point to the beginning of the thread's privileged stack area, optionally reprogramming the PSPLIM if stack limit checking is enabled
- modify CONTROL register to switch to privileged mode
- modify the return address in the SVC exception stack frame, so that after exception return the system call dispatcher is executed (in thread privileged mode)

Once the system call execution is completed the system call dispatcher will restore the user's original PSP and PSPLIM and switch the CONTROL register back to unprivileged mode before returning back to the caller of the system call.

System calls execute in thread mode and can be preempted by interrupts at any time. A thread may also be context-switched-out while doing a system call; the system call will resume as soon as the thread is switched-in again.

The system call dispatcher executes at SVC priority, therefore it cannot be preempted by HW interrupts (with the exception of ZLIs), which may observe some additional interrupt latency if they occur during a system call preparation.

MPU-assisted stack overflow detection Cortex-M platforms with MPU may enable `CONFIG_MPU_STACK_GUARD` to enable the MPU-based stack overflow detection mechanism. The following points need to be considered when enabling the MPU stack guards

- stack overflows are triggering processor faults as soon as they occur
- the mechanism is essential for detecting stack overflows in supervisor threads, or user threads in privileged mode; stack overflows in threads in user mode will always be detected regardless of `CONFIG_MPU_STACK_GUARD` being set.
- stack overflows are always detected, however, the mechanism does not guarantee that no memory corruption occurs when supervisor threads overflow their stack memory
- `CONFIG_MPU_STACK_GUARD` will normally reserve one MPU region for programming the stack guard (in certain Arm v8-M configurations with `CONFIG_MPU_GAP_FILLING` enabled 2 MPU regions are required to implement the guard feature)
- MPU guards are re-programmed at every context-switch, adding a small overhead to the thread swap routine. Compared, however, to the `CONFIG_BUILTIN_STACK_GUARD` feature, no re-programming occurs during system calls.
- When `CONFIG_HW_STACK_PROTECTION` is enabled on Arm v8-M platforms the native stack limit checking mechanism is used by default instead of the MPU-based stack overflow detection mechanism; users may override this setting by manually enabling `CONFIG_MPU_STACK_GUARD` in these scenarios.

Memory map and MPU considerations

Fixed MPU regions By default, when `CONFIG_ARM_MPU` is enabled a set of *fixed* MPU regions are programmed during system boot.

- One MPU region programs the entire flash area as read-execute. User can override this setting by enabling `CONFIG_MPU_ALLOW_FLASH_WRITE`, which programs the flash with RWX permissions. If `CONFIG_USERSPACE` is enabled unprivileged access on the entire flash area is allowed.
- One MPU region programs the entire SRAM area with privileged-only RW permissions. That is, an MPU region is utilized to disallow execute permissions on SRAM. (An exception to this setting is when `CONFIG_MPU_GAP_FILLING` is disabled (Arm v8-M only); in that case no SRAM MPU programming is done so the access is determined by the default Arm memory map policies, allowing for privileged-only RWX permissions on SRAM).

The above MPU regions are defined in `soc/arm/common/arm_mpu_regions.c`. Alternative MPU configurations are allowed by enabling `CONFIG_CPU_HAS_CUSTOM_FIXED_SOC_MPU_REGIONS`. When enabled, this option signifies that the Cortex-M SoC will define and configure its own fixed MPU regions in the SoC definition.

Static MPU regions Additional *static* MPU regions may be programmed once during system boot. These regions are required to enable certain features

- a RX region to allow execution from SRAM, when `CONFIG_ARCH_HAS_RAMFUNC_SUPPORT` is enabled and users have defined functions to execute from SRAM.
- a RX region for relocating text sections to SRAM, when `CONFIG_CODE_DATA_RELOCATION_SRAM` is enabled
- a no-cache region to allow for a none-cacheable SRAM area, when `CONFIG_NOCACHE_MEMORY` is enabled
- a possibly unprivileged RW region for GCOV code coverage accounting area, when `CONFIG_COVERAGE_GCOV` is enabled
- a no-access region to implement null pointer dereference detection, when `CONFIG_NULL_POINTER_EXCEPTION_DETECTION_MPU` is enabled

The boundaries of these static MPU regions are derived from symbols exposed by the linker, in `include/linker/linker-defs.h`.

Dynamic MPU regions Certain thread-specific MPU regions may be re-programmed dynamically, at each thread context switch:

- an unprivileged RW region for the current thread's stack area (for user threads)
- a read-only region for the MPU stack guard
- unprivileged RW regions for the partitions of the currentl thread's application memory domain.

Considerations The number of available MPU regions for a Cortex-M platform is a limited resource. Most platforms have 8 MPU regions, while some Cortex-M33 or Cortex-M7 platforms may have up to 16 MPU regions. Therefore there is a relatively strict limitation on how many fixed, static and dynamic MPU regions may be programmed simultaneously. For platforms with 8 available MPU regions it might not be possible to enable all the aforementioned features that require MPU region programming. In most practical applications, however, only a certain set of features is required and 8 MPU regions are, in many cases, sufficient.

In Arm v8-M processors the MPU architecture does not allow programmed MPU regions to overlap. `CONFIG_MPU_GAP_FILLING` controls whether the fixed MPU region covering the entire SRAM is programmed. When it does, a full SRAM area partitioning is required, in order to program the static and the dynamic MPU regions. This increases the total number of required MPU regions. When `CONFIG_MPU_GAP_FILLING` is not enabled the fixed MPU region convering the entire SRAM is not programmed, thus, the static and dynamic regions are simply programmed on top of the always-existing background region (full-SRAM partitioning is not required). Note, however, that the background SRAM region allows execution from SRAM, so when `CONFIG_MPU_GAP_FILLING` is not set Zephyr is not protected against attacks that attempt to execute malicious code from SRAM.

Floating point Services Both unshared and shared FP registers mode are supported in Cortex-M (see *Floating Point Services* for more details).

When FPU support is enabled in the build (`CONFIG_FPU` is enabled), the sharing FP registers mode (`CONFIG_FPU_SHARING`) is enabled by default. This is done as some compiler configurations may activate a floating point context by generating FP instructions for any thread, regardless of whether floating point calculations are performed, and that context must be preserved when switching such threads in and out.

The developers can still disable the FP sharing mode in their application projects, and switch to Unshared FP registers mode, if it is guaranteed that the image code does not generate FP instructions outside the single thread context that is allowed (and supposed) to do so.

Under FPU sharing mode, the callee-saved FPU registers are saved and restored in context-switch, if the corresponding threads have an active FP context. This adds some runtime overhead on the swap routine. In addition to the runtime overhead, the sharing FPU mode

- requires additional memory for each thread to save the callee-saved FP registers
- requires additional stack memory for each thread, to stack the caller-saved FP registers, upon exception entry, if an FP context is active. Note, however, that since lazy stacking is enabled, there is no runtime overhead of FP context stacking in regular interrupts (FP state preservation is only activated in the swap routine in PendSV interrupt).

Misc

Chain-loadable images Cortex-M applications may either be standalone images or chain-loadable, for instance, by a bootloader. Application images chain-loadable by bootloaders (or other applications) normally occupy a specific area in the flash denoted as their *code partition*. `CONFIG_USE_DT_CODE_PARTITION` will ensure that a Zephyr chain-loadable image will be linked into its code partition, specified in Device-Tree.

HW initialization at boot In order to boot properly, chain-loaded applications may require that the core Arm hardware registers and peripherals are initialized in their reset values. Enabling `CONFIG_INIT_ARCH_HW_AT_BOOT` Zephyr to force the initialization of the internal Cortex-M architectural state during boot to the reset values as specified by the corresponding Arm architecture manual.

Software vector relaying In Cortex-M platforms that implement the VTOR register (see `CONFIG_CPU_CORTEX_M_HAS_VTOR`), chain-loadable images relocate the Cortex-M vector table by updating the VTOR register with the offset of the image vector table.

Baseline Cortex-M platforms without VTOR register might not be able to relocate their vector table which remains at a fixed location. Therefore, a chain-loadable image will require an alternative way to route HW interrupts and system exeptions to its own vector table; this is achieved with software vector relaying.

When a bootloader image enables `CONFIG_SW_VECTOR_RELAY` it is able to relay exceptions and interrupts based on a vector table pointer that is set by the chain-loadable application. The latter sets the `CONFIG_SW_VECTOR_RELAY_CLIENT` option to instruct the boot sequence to set the vector table pointer in SRAM so that the bootloader can forward the exceptions and interrupts to the chain-loadable image's software vector table.

While this feature is intended for processors without VTOR register, it may also be used in Mainline Cortex-M platforms.

Code relocation Cortex-M support the code relocation feature. When `CONFIG_CODE_DATA_RELOCATION_SRAM` is selected, Zephyr will relocate .text, data and .bss sections from the specified files and place it in SRAM. It is possible to relocate only parts of the code sections into SRAM, without relocating the whole image text and data sections. More details on the code relocation feature can be found in *Code And Data Relocation*.

Linking Cortex-M applications

Most Cortex-M platforms make use of the default Cortex-M GCC linker script in `include/arch/arm/aarch32/cortex-m/scripts/linked.ld`, although it is possible for platforms to use a custom linker script as well.

CMSIS

Cortex-M CMSIS headers are hosted in a standalone module repository: zephyrproject-rtos/cmsis.

`CONFIG_CPU_CORTEX_M` selects `CONFIG_HAS_CMSIS_CORE` to signify that CMSIS headers are available for all supported Cortex-M variants.

Testing

A list of unit tests for the Cortex-M porting and miscellaneous features is present in `tests/arch/arm/`. The tests suites are continuously extended and new test suites are added, in an effort to increase the coverage of the Cortex-M architecture support in Zephyr.

QEMU

We use QEMU to verify the implemented features of the Cortex-M architecture port in Zephyr. Adequate coverage is achieved by defining and utilizing a list of QEMU targets, each with a specific architecture variant and Arm peripheral support list.

The table below lists the QEMU platform targets defined in Zephyr along with the corresponding Cortex-M implementation variant and the peripherals these targets emulate.

	QEMU target				
Architecture variant	Arm v6-M	Arm v7-M		Arm v8-M	Arm v8.1-M
	qemu_cortex_m0	**qemu_cortex_m3**	**mps2_an385**	**mps2_an521**	**mps3_an547**
Emulated features					
NVIC	Y	Y	Y	Y	Y
BASEPRI	N	Y	Y	Y	Y
SysTick	N	Y	Y	Y	Y
MPU	N	N	Y	Y	Y
FPU	N	N	N	Y	N
SPLIM	N	N	N	Y	Y
TrustZone-M	N	N	N	Y	N

Maintainers & Collaborators

The status of the Arm Cortex-M architecture port in Zephyr is: *maintained*. The updated list of maintainers and collaborators for Cortex-M can be found in `MAINTAINERS.yml`.

8.2.3 x86 Developer Guide

Overview

This page contains information on certain aspects when developing for x86-based platforms.

Virtual Memory

During very early boot, page tables are loaded so technically the kernel is executing in virtual address space. By default, physical and virtual memory are identity mapped and thus giving the appearance of execution taking place in physical address space. The physical address space is marked by kconfig `CONFIG_SRAM_BASE_ADDRESS` and `CONFIG_SRAM_SIZE` while the virtual address space is marked by

CONFIG_KERNEL_VM_BASE and CONFIG_KERNEL_VM_SIZE. Note that CONFIG_SRAM_OFFSET controls where the Zephyr kernel is being placed in the memory, and its counterpart CONFIG_KERNEL_VM_OFFSET.

Separate Virtual Address Space from Physical Address Space On 32-bit x86, it is possible to have separate phyiscal and virtual address space. Code and data are linked in virtual address space, but are still loaded in physical memory. However, during boot, code and data must be available and also addressable in physical address space before vm_enter inside arch/x86/core/ia32/crt0.S. After vm_enter, code execution is done via virtual addresses and data can be referred via their virtual addresses. This is possible as the page table generation script (arch/x86/gen_mmu.py) identity maps the physical addresses at the page directory level, in addition to mapping virtual addresses to the physical memory. Later in the boot process, the entries for identity mapping at the page directory level are cleared in z_x86_mmu_init(), effectively removing the identity mapping of physical memory. This unmapping must be done for userspace isolation or else they would be able to access restricted memory via physical addresses. Since the identity mapping is done at the page directory level, there is no need to allocate additional space for the page table. However, additional space may still be required for additional page directory table.

There are restrictions on where virtual address space can be:

- Physical and virtual address spaces must be disjoint. This is required as the entries in page directory table will be cleared. If they are not disjoint, it would clear the entries needed for virtual addresses.
  - If CONFIG_X86_PAE is enabled (=y), each address space must reside in their own 1GB region, due to each entry of PDP (Page Directory Pointer) covers 1GB of memory. For example:
    * Assuming CONFIG_SRAM_OFFSET and CONFIG_KERNEL_VM_OFFSET are both 0x0.
    * CONFIG_SRAM_BASE_ADDRESS == 0x00000000 and CONFIG_KERNEL_VM_BASE = 0x40000000 is valid, while
    * CONFIG_SRAM_BASE_ADDRESS == 0x00000000 and CONFIG_KERNEL_VM_BASE = 0x20000000 is not.
  - If CONFIG_X86_PAE is disabled (=n), each address space must reside in their own 4MB region, due to each entry of PD (Page Directory) covers 4MB of memory.
  - Both CONFIG_SRAM_BASE_ADDRESS and CONFIG_KERNEL_VM_BASE must also align with the starting addresses of targeted regions.

Specifying Additional Memory Mappings at Build Time

The page table generation script (arch/x86/gen_mmu.py) generates the necessary multi-level page tables for code execution and data access using the kernel image produced by the first linker pass. Additional command line arguments can be passed to the script to generate additional memory mappings. This is useful for static mappings and/or device MMIO access during very early boot. To pass extra command line arguments to the script, populate a CMake list named X86_EXTRA_GEN_MMU_ARGUMENTS in the board configuration file. Here is an example:

```
set(X86_EXTRA_GEN_MMU_ARGUMENTS
    --map 0xA0000000,0x2000
    --map 0x80000000,0x400000,LWUX,0xB0000000)
```

The argument --map takes the following value: <physical address>,<size>[,<flags:LUWX>[,<virtual adderss>]], where:

- <physical address> is the physical address of the mapping. (Required)
- <size> is the size of the region to be mapped. (Required)
- <flags> is the flag associated with the mapping: (Optional)
  - L: Large page at the page directory level.

  - `U`: Allow userspace access.
  - `W`: Read/write.
  - `X`: Allow execution.
  - `D`: Cache disabled.
    * Default is small page (4KB), supervisor only, read only, and execution disabled.
- `<virtual address` is the virtual address of the mapping. (Optional)

Note that specifying additional memory mappings requires larger storage space for the pre-allocated page tables (both kernel and per-domain tables). `CONFIG_X86_EXTRA_PAGE_TABLE_PAGES` is needed to specify how many more memory pages to be reserved for the page tables. If the needed space is not exactly the same as required space, the `gen_mmu.py` script will print out a message indicating what needs to be the value for the kconfig.

8.3 Bluetooth

This section contains information regarding the Bluetooth stack of the Zephyr OS. You can use this information to understand the principles behind the operation of the layers and how they were implemented.

Zephyr includes a complete Bluetooth Low Energy stack from application to radio hardware, as well as portions of a Classical Bluetooth (BR/EDR) Host layer.

8.3.1 Overview

Since its inception, Zephyr has had a strong focus on Bluetooth and, in particular, on Bluetooth Low Energy (BLE). Through the contributions of several companies and individuals involved in existing open source implementations of the Bluetooth specification (Linux's BlueZ) as well as the design and development of BLE radio hardware, the protocol stack in Zephyr has grown to be mature and feature-rich, as can be seen in the section below.

Supported Features

Zephyr comes integrated with a feature-rich and highly configurable Bluetooth stack.

- Bluetooth 5.0 compliant (ESR10)
  - Highly configurable
    * Features, buffer sizes/counts, stack sizes, etc.
  - Portable to all architectures supported by Zephyr (including big and little endian, alignment flavors and more)
  - Support for all combinations of Host and Controller builds:
    * Controller-only (HCI) over UART, SPI, and USB physical transports
    * Host-only over UART, SPI, and IPM (shared memory)
    * Combined (Host + Controller)
- Bluetooth-SIG qualified
  - Controller on Nordic Semiconductor hardware
  - Conformance tests run regularly on all layers

- Bluetooth Low Energy Controller support (LE Link Layer)
  - Unlimited role and connection count, all roles supported
  - Concurrent multi-protocol support ready
  - Intelligent scheduling of roles to minimize overlap
  - Portable design to any open BLE radio, currently supports Nordic Semiconductor nRF51 and nRF52, as well as proprietary radios
  - Supports little and big endian architectures, and abstracts the hard real-time specifics so that they can be encapsulated in a hardware-specific module
  - Support for Controller (HCI) builds over different physical transports
- Bluetooth Host support
  - Generic Access Profile (GAP) with all possible LE roles
    - Peripheral & Central
    - Observer & Broadcaster
  - GATT (Generic Attribute Profile)
    - Server (to be a sensor)
    - Client (to connect to sensors)
  - Pairing support, including the Secure Connections feature from Bluetooth 4.2
  - Non-volatile storage support for permanent storage of Bluetooth-specific settings and data
  - Bluetooth mesh support
    - Relay, Friend Node, Low-Power Node (LPN) and GATT Proxy features
    - Both Provisioning bearers supported (PB-ADV & PB-GATT)
    - Highly configurable, fits as small as 16k RAM devices
  - IPSP/6LoWPAN for IPv6 connectivity over Bluetooth LE
    - IPSP node sample application
  - Basic Bluetooth BR/EDR (Classic) support
    - Generic Access Profile (GAP)
    - Logical Link Control and Adaptation Protocol (L2CAP)
    - Serial Port emulation (RFCOMM protocol)
    - Service Discovery Protocol (SDP)
  - Clean HCI driver abstraction
    - 3-Wire (H:5) & 5-Wire (H:4) UART
    - SPI
    - Local controller support as a virtual HCI driver
  - Verified with multiple popular controllers

8.3.2 Bluetooth Stack Architecture

Overview

This page describes the software architecture of Zephyr's Bluetooth protocol stack.

Note: Zephyr supports mainly Bluetooth Low Energy (BLE), the low-power version of the Bluetooth specification. Zephyr also has limited support for portions of the BR/EDR Host. Throughout this architecture document we use BLE interchangeably for Bluetooth except when noted.

BLE Layers There are 3 main layers that together constitute a full Bluetooth Low Energy protocol stack:

- **Host**: This layer sits right below the application, and is comprised of multiple (non real-time) network and transport protocols enabling applications to communicate with peer devices in a standard and interoperable way.
- **Controller**: The Controller implements the Link Layer (LE LL), the low-level, real-time protocol which provides, in conjunction with the Radio Hardware, standard interoperable over the air communication. The LL schedules packet reception and transmission, guarantees the delivery of data, and handles all the LL control procedures.
- **Radio Hardware**: Hardware implements the required analog and digital baseband functional blocks that permit the Link Layer firmware to send and receive in the 2.4GHz band of the spectrum.

Host Controller Interface The Bluetooth Specification describes the format in which a Host must communicate with a Controller. This is called the Host Controller Interface (HCI) protocol. HCI can be implemented over a range of different physical transports like UART, SPI, or USB. This protocol defines the commands that a Host can send to a Controller and the events that it can expect in return, and also the format for user and protocol data that needs to go over the air. The HCI ensures that different Host and Controller implementations can communicate in a standard way making it possible to combine Hosts and Controllers from different vendors.

Configurations The three separate layers of the protocol and the standardized interface make it possible to implement the Host and Controller on different platforms. The two following configurations are commonly used:

- **Single-chip configuration**: In this configuration, a single microcontroller implements all three layers and the application itself. This can also be called a system-on-chip (SoC) implementation. In this case the BLE Host and the BLE Controller communicate directly through function calls and queues in RAM. The Bluetooth specification does not specify how HCI is implemented in this single-chip configuration and so how HCI commands, events, and data flows between the two can be implementation-specific. This configuration is well suited for those applications and designs that require a small footprint and the lowest possible power consumption, since everything runs on a single IC.
- **Dual-chip configuration**: This configuration uses two separate ICs, one running the Application and the Host, and a second one with the Controller and the Radio Hardware. This is sometimes also called a connectivity-chip configuration. This configuration allows for a wider variety of combinations of Hosts when using the Zephyr OS as a Controller. Since HCI ensures interoperability among Host and Controller implementations, including of course Zephyr's very own BLE Host and Controller, users of the Zephyr Controller can choose to use whatever Host running on any platform they prefer. For example, the host can be the Linux BLE Host stack (BlueZ) running on any processor capable of supporting Linux. The Host processor may of course also run Zephyr and the Zephyr OS BLE Host. Conversely, combining an IC running the Zephyr Host with an external Controller that does not run Zephyr is also supported.

Build Types The Zephyr software stack as an RTOS is highly configurable, and in particular, the BLE subsystem can be configured in multiple ways during the build process to include only the features and layers that are required to reduce RAM and ROM footprint as well as power consumption. Here's a short list of the different BLE-enabled builds that can be produced from the Zephyr project codebase:

- **Controller-only build**: When built as a BLE Controller, Zephyr includes the Link Layer and a special application. This application is different depending on the physical transport chosen for HCI:
  - hci_uart
  - hci_usb
  - hci_spi

  This application acts as a bridge between the UART, SPI or USB peripherals and the Controller subsystem, listening for HCI commands, sending application data and responding with events and received data. A build of this type sets the following Kconfig option values:
  - `CONFIG_BT =y`
  - `CONFIG_BT_HCI =y`
  - `CONFIG_BT_HCI_RAW =y`
  - `CONFIG_BT_CTLR =y`
  - `CONFIG_BT_LL_SW_SPLIT =y` (if using the open source Link Layer)
- **Host-only build**: A Zephyr OS Host build will contain the Application and the BLE Host, along with an HCI driver (UART or SPI) to interface with an external Controller chip. A build of this type sets the following Kconfig option values:
  - `CONFIG_BT =y`
  - `CONFIG_BT_HCI =y`
  - `CONFIG_BT_CTLR =n`

  All of the samples located in `samples/bluetooth` except for the ones used for Controller-only builds can be built as Host-only
- **Combined build**: This includes the Application, the Host and the Controller, and it is used exclusively for single-chip (SoC) configurations. A build of this type sets the following Kconfig option values:
  - `CONFIG_BT =y`
  - `CONFIG_BT_HCI =y`
  - `CONFIG_BT_CTLR =y`
  - `CONFIG_BT_LL_SW_SPLIT =y` (if using the open source Link Layer)

  All of the samples located in `samples/bluetooth` except for the ones used for Controller-only builds can be built as Combined

The picture below shows the SoC or single-chip configuration when using a Zephyr combined build (a build that includes both a BLE Host and a Controller in the same firmware image that is programmed onto the chip):

When using connectivity or dual-chip configurations, several Host and Controller combinations are possible, some of which are depicted below:

When using a Zephyr Host (left side of image), two instances of Zephyr OS must be built with different configurations, yielding two separate images that must be programmed into each of the chips respectively. The Host build image contains the application, the BLE Host and the selected HCI driver (UART or SPI), while the Controller build runs either the hci_uart, or the hci_spi app to provide an interface to the BLE Controller.

This configuration is not limited to using a Zephyr OS Host, as the right side of the image shows. One can indeed take one of the many existing GNU/Linux distributions, most of which include Linux's own BLE Host (BlueZ), to connect it via UART or USB to one or more instances of the Zephyr OS Controller build. BlueZ as a Host supports multiple Controllers simultaneously for applications that require more than one BLE radio operating at the same time but sharing the same Host stack.

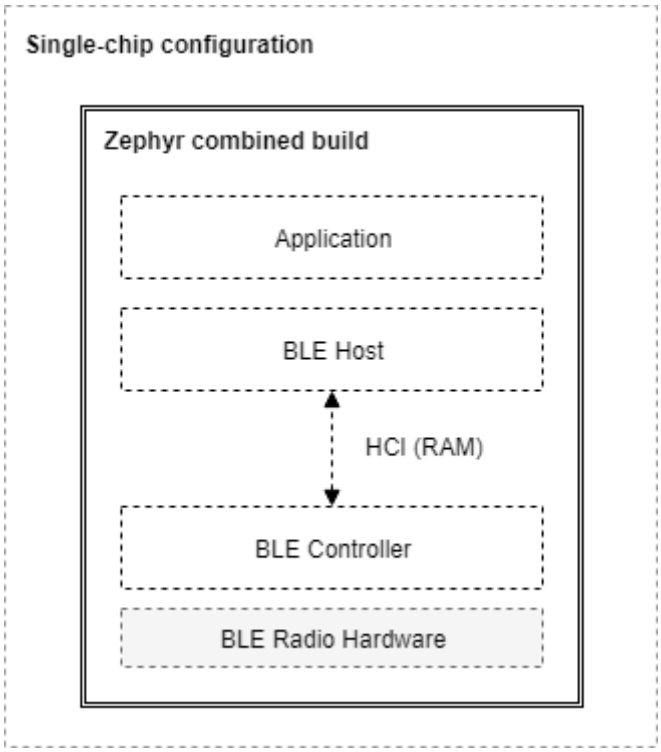


Fig. 1: A Combined build on a Single-Chip configuration

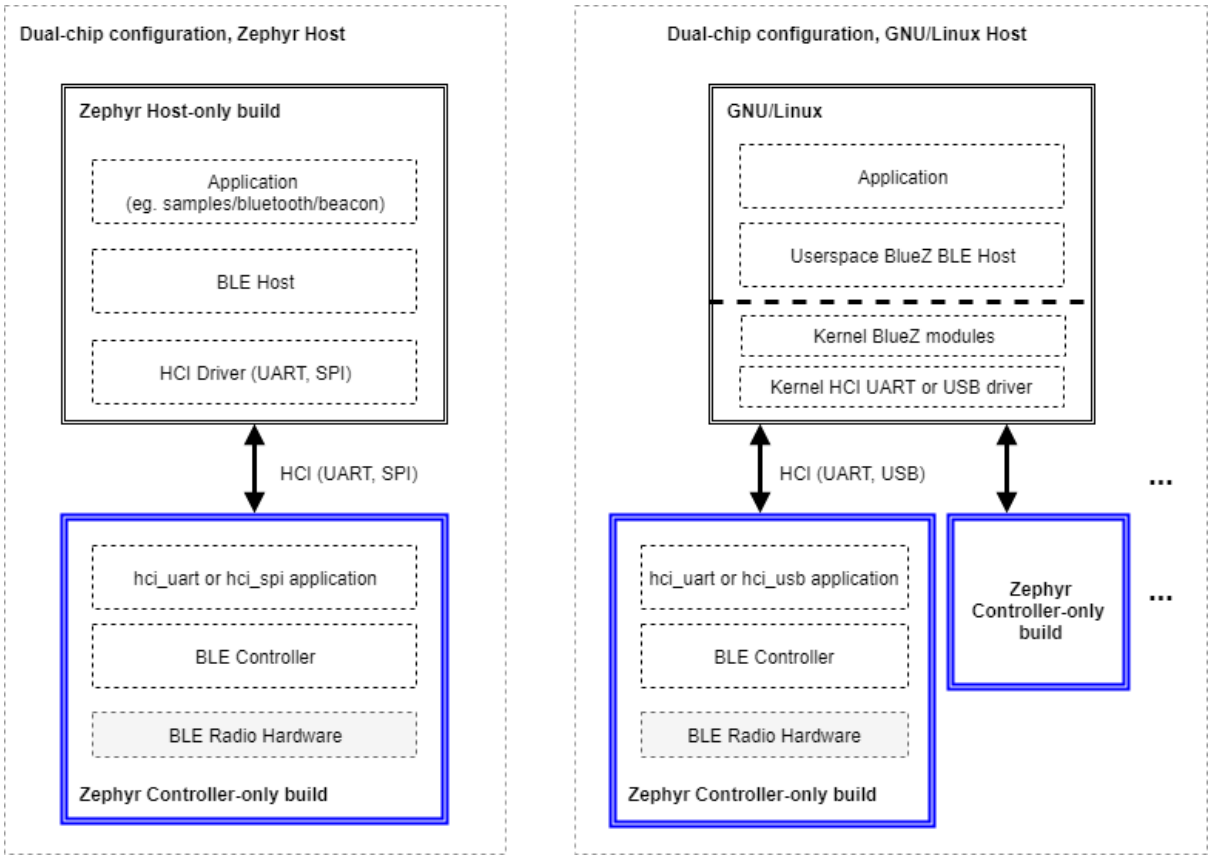


Fig. 2: Host-only and Controller-only builds on dual-chip configurations

Source tree layout

The stack is split up as follows in the source tree:

`subsys/bluetooth/host` The host stack. This is where the HCI command and event handling as well as connection tracking happens. The implementation of the core protocols such as L2CAP, ATT, and SMP is also here.

`subsys/bluetooth/controller` Bluetooth Controller implementation. Implements the controller-side of HCI, the Link Layer as well as access to the radio transceiver.

`include/bluetooth/` Public API header files. These are the header files applications need to include in order to use Bluetooth functionality.

`drivers/bluetooth/` HCI transport drivers. Every HCI transport needs its own driver. For example, the two common types of UART transport protocols (3-Wire and 5-Wire) have their own drivers.

`samples/bluetooth/` Sample Bluetooth code. This is a good reference to get started with Bluetooth application development.

`tests/bluetooth/` Test applications. These applications are used to verify the functionality of the Bluetooth stack, but are not necessary the best source for sample code (see `samples/bluetooth` instead).

`doc/guides/bluetooth/` Extra documentation, such as PICS documents.

Host

The Bluetooth Host implements all the higher-level protocols and profiles, and most importantly, provides a high-level API for applications. The following diagram depicts the main protocol & profile layers of the host.

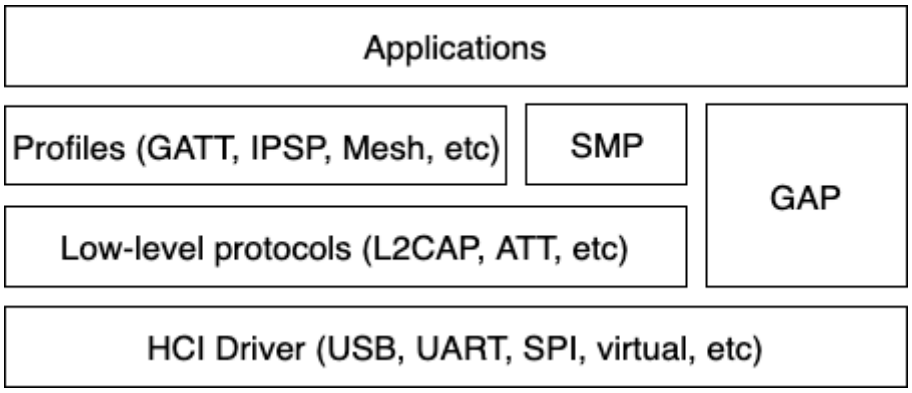


Fig. 3: Bluetooth Host protocol & profile layers.

Lowest down in the host stack sits a so-called HCI driver, which is responsible for abstracting away the details of the HCI transport. It provides a basic API for delivering data from the controller to the host, and vice-versa.

Perhaps the most important block above the HCI handling is the Generic Access Profile (GAP). GAP simplifies Bluetooth LE access by defining four distinct roles of BLE usage:

- Connection-oriented roles
  - Peripheral (e.g. a smart sensor, often with a limited user interface)
  - Central (typically a mobile phone or a PC)
- Connection-less roles
  - Broadcaster (sending out BLE advertisements, e.g. a smart beacon)

- Observer (scanning for BLE advertisements)

Each role comes with its own build-time configuration option: `CONFIG_BT_PERIPHERAL`, `CONFIG_BT_CENTRAL`, `CONFIG_BT_BROADCASTER` & `CONFIG_BT_OBSERVER`. Of the connection-oriented roles central implicitly enables observer role, and peripheral implicitly enables broadcaster role. Usually the first step when creating an application is to decide which roles are needed and go from there. Bluetooth mesh is a slightly special case, requiring at least the observer and broadcaster roles, and possibly also the Peripheral role. This will be described in more detail in a later section.

Peripheral role Most Zephyr-based BLE devices will most likely be peripheral-role devices. This means that they perform connectable advertising and expose one or more GATT services. After registering services using the *bt_gatt_service_register()* API the application will typically start connectable advertising using the *bt_le_adv_start()* API.

There are several peripheral sample applications available in the tree, such as samples/bluetooth/peripheral_hr.

Central role Central role may not be as common for Zephyr-based devices as peripheral role, but it is still a plausible one and equally well supported in Zephyr. Rather than accepting connections from other devices a central role device will scan for available peripheral device and choose one to connect to. Once connected, a central will typically act as a GATT client, first performing discovery of available services and then accessing one or more supported services.

To initially discover a device to connect to the application will likely use the *bt_le_scan_start()* API, wait for an appropriate device to be found (using the scan callback), stop scanning using *bt_le_scan_stop()* and then connect to the device using `bt_conn_create_le()`. If the central wants to keep automatically reconnecting to the peripheral it should use the *bt_le_set_auto_conn()* API.

There are some sample applications for the central role available in the tree, such as samples/bluetooth/central_hr.

Observer role An observer role device will use the *bt_le_scan_start()* API to scan for device, but it will not connect to any of them. Instead it will simply utilize the advertising data of found devices, combining it optionally with the received signal strength (RSSI).

Broadcaster role A broadcaster role device will use the *bt_le_adv_start()* API to advertise specific advertising data, but the type of advertising will be non-connectable, i.e. other device will not be able to connect to it.

Connections Connection handling and the related APIs can be found in the *Connection Management* section.

Security To achieve a secure relationship between two Bluetooth devices a process called pairing is used. This process can either be triggered implicitly through the security properties of GATT services, or explicitly using the `bt_conn_security()` API on a connection object.

To achieve a higher security level, and protect against Man-In-The-Middle (MITM) attacks, it is recommended to use some out-of-band channel during the pairing. If the devices have a sufficient user interface this "channel" is the user itself. The capabilities of the device are registered using the *bt_conn_auth_cb_register()* API. The *bt_conn_auth_cb* struct that's passed to this API has a set of optional callbacks that can be used during the pairing - if the device lacks some feature the corresponding callback may be set to NULL. For example, if the device does not have an input method but does have a display, the `passkey_entry` and `passkey_confirm` callbacks would be set to NULL, but the `passkey_display` would be set to a callback capable of displaying a passkey to the user.

Depending on the local and remote security requirements & capabilities, there are four possible security levels that can be reached:

`BT_SECURITY_L1` No encryption and no authentication.

`BT_SECURITY_L2` Encryption but no authentication (no MITM protection).

`BT_SECURITY_L3` Encryption and authentication using the legacy pairing method from Bluetooth 4.0 and 4.1.

`BT_SECURITY_L4` Encryption and authentication using the LE Secure Connections feature available since Bluetooth 4.2.

Note: Mesh has its own security solution through a process called provisioning. It follows a similar procedure as pairing, but is done using separate mesh-specific APIs.

L2CAP L2CAP stands for the Logical Link Control and Adaptation Protocol. It is a common layer for all communication over Bluetooth connections, however an application comes in direct contact with it only when using it in the so-called Connection-oriented Channels (CoC) mode. More information on this can be found in the *L2CAP API section*.

GATT The Generic Attribute Profile is the most common means of communication over LE connections. A more detailed description of this layer and the API reference can be found in the *GATT API reference section*.

Mesh Mesh is a little bit special when it comes to the needed GAP roles. By default, mesh requires both observer and broadcaster role to be enabled. If the optional GATT Proxy feature is desired, then peripheral role should also be enabled.

The API reference for mesh can be found in the *Mesh API reference section*.

Persistent storage The Bluetooth host stack uses the settings subsystem to implement persistent storage to flash. This requires the presence of a flash driver and a designated "storage" partition on flash. A typical set of configuration options needed will look something like the following:

```
CONFIG_BT_SETTINGS=y
CONFIG_FLASH=y
CONFIG_FLASH_PAGE_LAYOUT=y
CONFIG_FLASH_MAP=y
CONFIG_NVS=y
CONFIG_SETTINGS=y
```

Once enabled, it is the responsibility of the application to call settings_load() after having initialized Bluetooth (using the bt_enable() API).

BLE Controller

Standard

Split

8.3.3 Bluetooth Qualification

Qualification Listings

The Zephyr BLE stack has obtained qualification listings for both the Host and the Controller. See the tables below for a list of qualification listings

Host qualifications

Zephyr version	Link	Qualifying Company
2.2.x	QDID 151074	Demant A/S
1.14.x	QDID 139258	The Linux Foundation
1.13	QDID 119517	Nordic Semiconductor

Mesh qualifications

Zephyr version	Link	Qualifying Company
1.14.x	QDID 139259	The Linux Foundation

Controller qualifications

Zephyr version	Link	Qualifying Company	Compatible Hardware
2.2.x	QDID 150092	Nordic Semiconductor	nRF52x
1.14.x	QDID 135679	Nordic Semiconductor	nRF52x
1.9 to 1.13	QDID 101395	Nordic Semiconductor	nRF52x

ICS Features

The ICS features for each supported protocol & profile can be found in the following documents:

GAP ICS PTS version: 8.0.3

M - mandatory

O - optional

Device Configuration

Parameter Name	Selected	Description
TSPC_GAP_0_1	False	BR/EDR (C.1)
TSPC_GAP_0_2	True	LE (C.2)
TSPC_GAP_0_3	False	BR/EDR/LE (C.3)

Modes

Parameter Name	Selected	Description
TSPC_GAP_1_1	False	Non-discoverable mode (C.1)
TSPC_GAP_1_2	False	Limited-discoverable mode (O)
TSPC_GAP_1_3	False	General-discoverable mode (O)
TSPC_GAP_1_4	False	Non-connectable mode (O)
TSPC_GAP_1_5	False	Connectable mode (M)
TSPC_GAP_1_6	False	Non-bondable mode (O)
TSPC_GAP_1_7	False	Bondable mode (C.2)
TSPC_GAP_1_8	False	Non-Synchronizable Mode (C.3)
TSPC_GAP_1_9	False	Synchronizable Mode (C.4)

Security Aspects

Parameter Name	Selected	Description
TSPC_GAP_2_1	False	Authentication procedure (C.1)
TSPC_GAP_2_2	False	Support of LMP-Authentication (M)
TSPC_GAP_2_3	False	Initiate LMP-Authentication (C.5)
TSPC_GAP_2_4	False	Security mode 1 (C.2)
TSPC_GAP_2_5	False	Security mode 2 (O)
TSPC_GAP_2_6	False	Security mode 3 (C.7)
TSPC_GAP_2_7	False	Security mode 4 (M)
TSPC_GAP_2_7a	False	Security mode 4, level 4 (C.9)
TSPC_GAP_2_7b	False	Security mode 4, level 3 (C.9)
TSPC_GAP_2_7c	False	Security mode 4, level 2 (C.9)
TSPC_GAP_2_7d	False	Security mode 4, level 1 (C.9)
TSPC_GAP_2_8	False	Support of Authenticated link key (C.6)
TSPC_GAP_2_9	False	Support of Unauthenticated link key (C.6)
TSPC_GAP_2_10	False	Security Optional (C.6)
TSPC_GAP_2_11	False	Secure Connections Only Mode (C.8)
TSPC_GAP_2_12	False	56-bit minimum encryption key size (C.10)
TSPC_GAP_2_13	False	128-bit encryption key size capable (C.11)

Idle Mode Procedures

Parameter Name	Selected	Description
TSPC_GAP_3_1	False	Initiation of general inquiry (C.1)
TSPC_GAP_3_2	False	Initiation of limited inquiry (C.1)
TSPC_GAP_3_3	False	Initiation of name discovery (O)
TSPC_GAP_3_4	False	Initiation of device discovery (O)
TSPC_GAP_3_5	False	Initiation of general bonding (O)
TSPC_GAP_3_6	False	Initiation of dedicated bonding (O)

Establishment Procedures

Parameter Name	Selected	Description
TSPC_GAP_4_1	False	Support link establishment as initiator (M)
TSPC_GAP_4_2	False	Support link establishment as acceptor (M)
TSPC_GAP_4_3	False	Support channel establishment as initiator (O)
TSPC_GAP_4_4	False	Support channel establishment as acceptor (M)
TSPC_GAP_4_5	False	Support connection establishment as initiator (O)
TSPC_GAP_4_6	False	Support connection establishment as acceptor (O)
TSPC_GAP_4_7	False	Support synchronization establishment as receiver (C.1)

LE Roles

Parameter Name	Selected	Description
TSPC_GAP_5_1	True	Broadcaster (C.1)
TSPC_GAP_5_2	True	Observer (C.1)
TSPC_GAP_5_3	True	Peripheral (C.1)
TSPC_GAP_5_4	True	Central (C.1)

Broadcaster Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_6_1	True	Transmitter (M)
TSPC_GAP_6_2	True	Receiver (O)

Broadcaster Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_7_1	True	Standby (M)
TSPC_GAP_7_2	True	Advertising (M)
TSPC_GAP_7_3	False	Isochronous Broadcasting State (C.1)

Broadcaster Link Layer Advertising Event Types

Parameter Name	Selected	Description
TSPC_GAP_8_1	True	Non-Connectable Undirected Event (M)
TSPC_GAP_8_2	True	Scannable Undirected Event (O)
TSPC_GAP_8_3	True	Non-Connectable and Non-Scannable Directed Event (C.1)
TSPC_GAP_8_4	True	Scannable Directed Event (C.1)

Broadcaster Link Layer Advertising Data Types

Parameter Name	Selected	Description
TSPC_GAP_8a_1	True	AD Type – Service UUID (O)
TSPC_GAP_8a_2	True	AD Type – Local Name (O)
TSPC_GAP_8a_3	True	AD Type – Flags (O)
TSPC_GAP_8a_4	True	AD Type – Manufacturer Specific Data (O)
TSPC_GAP_8a_5	True	AD Type – TX Power Level (O)
TSPC_GAP_8a_6	False	AD Type – Security Manager Out of Band (OOB) (C.1)
TSPC_GAP_8a_7	True	AD Type – Security Manager TK Value (O)
TSPC_GAP_8a_8	True	AD Type – Peripheral Connection Interval Range (O)
TSPC_GAP_8a_9	True	AD Type - Service Solicitation (O)
TSPC_GAP_8a_10	True	AD Type – Service Data (O)
TSPC_GAP_8a_11	True	AD Type – Appearance (O)
TSPC_GAP_8a_12	True	AD Type – Public Target Address (O)
TSPC_GAP_8a_13	True	AD Type – Random Target Address (O)
TSPC_GAP_8a_14	True	AD Type – Advertising Interval (O)
TSPC_GAP_8a_15	True	AD Type – LE Bluetooth Device Address (O)
TSPC_GAP_8a_16	True	AD Type – LE Role (O)
TSPC_GAP_8a_17	True	AD Type - URI (O)

Broadcaster Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_9_1	True	Non-Connectable Mode (M)

Broadcaster Broadcasting and Observing Features

Parameter Name	Selected	Description
TSPC_GAP_10_1	True	Broadcast Mode (M)
TSPC_GAP_10_2	False	Broadcast Isochronous Synchronizability mode (C.1)
TSPC_GAP_10_3	False	Broadcast Isochronous Broadcasting mode (C.2)
TSPC_GAP_10_4	False	Broadcast Isochronous Terminate procedure (C.1)
TSPC_GAP_10_5	False	Broadcast Isochronous Channel Map Update Procedure (C.1)

Broadcaster Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_11_1	True	Privacy Feature (O)
TSPC_GAP_11_2	True	Resolvable Private Address Generation Procedure (C.1)
TSPC_GAP_11_3	True	Non-Resolvable Private Address Generation Procedure (C.2)

Periodic Advertising Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_11a_1	False	Periodic Advertising Synchronizability mode (C.1)
TSPC_GAP_11a_2	False	Periodic Advertising mode (C.2)

Broadcaster Security Aspects Features

Parameter Name	Selected	Description
TSPC_GAP_11b_1	False	LE Security Mode 3 (C.1)
TSPC_GAP_11b_2	False	LE Security Mode 3, Level 1 (C.2)
TSPC_GAP_11b_3	False	LE Security Mode 3, Level 2 (C.2)
TSPC_GAP_11b_4	False	LE Security Mode 3, Level 3 (C.2)

Observer Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_12_1	True	Receiver (M)
TSPC_GAP_12_2	True	Transmitter (O)

Observer Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_13_1	True	Standby (M)
TSPC_GAP_13_2	True	Scanning (M)

Observer Link Layer Scanning Types

Parameter Name	Selected	Description
TSPC_GAP_14_1	True	Passive Scanning (M)
TSPC_GAP_14_2	True	Active Scanning (O)

Observer Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_15_1	True	Non-Connectable Modes (M)

Observer Broadcasting and Observing Features

Parameter Name	Selected	Description
TSPC_GAP_16_1	True	Observation Procedure (M)
TSPC_GAP_16_2	False	Broadcast Isochronous Synchronization Establishment procedure (C.1)
TSPC_GAP_16_3	False	Broadcast Isochronous Termination procedure (C.2)
TSPC_GAP_16_4	False	Broadcast Isochronous Channel Map Update Procedure (C.2)

Observer Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_17_1	True	Privacy Feature (O)
TSPC_GAP_17_2	True	Non-Resolvable Private Address Generation Procedure (C.1)
TSPC_GAP_17_3	True	Resolvable Private Address Resolution Procedure (O)
TSPC_GAP_17_4	True	Resolvable Private Address Generation Procedure (C.2)

Periodic Advertising Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_17a_1	False	Periodic Advertising Synchronization Establishment procedure without listening for periodic advertising (C.1)
TSPC_GAP_17a_2	False	Periodic Advertising Synchronization Establishment procedure with listening for periodic advertising (C.1)

Observer Security Aspects Features

Parameter Name	Selected	Description
TSPC_GAP_17b_1	False	LE Security Mode 3 (C.1)
TSPC_GAP_17b_2	False	LE Security Mode 3, Level 1 (C.2)
TSPC_GAP_17b_3	False	LE Security Mode 3, Level 2 (C.2)
TSPC_GAP_17b_4	False	LE Security Mode 3, Level 3 (C.2)

Peripheral Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_18_1	True	Transmitter (M)
TSPC_GAP_18_2	True	Receiver (M)

Peripheral Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_19_1	True	Standby (M)
TSPC_GAP_19_2	True	Advertising (M)
TSPC_GAP_19_3	True	Connection, Peripheral Role (M)

Peripheral Link Layer Advertising Event Types

Parameter Name	Selected	Description
TSPC_GAP_20_1	True	Connectable and Scannable Undirected Event (M)
TSPC_GAP_20_2	True	Connectable Directed Event (O)
TSPC_GAP_20_3	True	Non-Connectable and Non-Scannable Undirected Event (O)
TSPC_GAP_20_4	True	Scannable Undirected Event (O)
TSPC_GAP_20_5	True	Connectable Undirected Event (C.1)
TSPC_GAP_20_6	True	Non-Connectable and Non-Scannable Directed Event (C.1)
TSPC_GAP_20_7	True	Scannable Directed Event (C.1)

Peripheral Link Layer Advertising Data Types

Parameter Name	Selected	Description
TSPC_GAP_20A_1	True	AD Type – Service UUID (C.1)
TSPC_GAP_20A_2	True	AD Type – Local Name (C.1)
TSPC_GAP_20A_3	True	AD Type – Flags (C.2)
TSPC_GAP_20A_4	True	AD Type – Manufacturer Specific Data (C.1)
TSPC_GAP_20A_5	True	AD Type – TX Power Level (C.1)
TSPC_GAP_20A_6	False	AD Type – Security Manager Out of Band (OOB) (C.3)
TSPC_GAP_20A_7	True	AD Type – Security Manager TK Value (C.1)
TSPC_GAP_20A_8	True	AD Type – Peripheral Connection Interval Range (C.1)
TSPC_GAP_20A_9	True	AD Type – Service Solicitation (C.1)
TSPC_GAP_20A_10	True	AD Type – Service Data (C.1)
TSPC_GAP_20A_11	True	AD Type – Appearance (C.1)
TSPC_GAP_20A_12	True	AD Type – Public Target Address (C.1)
TSPC_GAP_20A_13	True	AD Type – Random Target Address (C.1)
TSPC_GAP_20A_14	True	AD Type – Advertising Interval (C.1)
TSPC_GAP_20A_15	True	AD Type – LE Bluetooth Device Address (C.1)
TSPC_GAP_20A_16	True	AD Type – LE Role (C.1)
TSPC_GAP_20A_17	True	AD Type – URI (O)

Peripheral Link Layer Control Procedures

Parameter Name	Selected	Description
TSPC_GAP_21_1	True	Connection Update Procedure (M)
TSPC_GAP_21_2	True	Channel Map Update Procedure (M)
TSPC_GAP_21_3	True	Encryption Procedure (O)
TSPC_GAP_21_4	True	Central Initiated Feature Exchange Procedure (M)
TSPC_GAP_21_5	True	Version Exchange Procedure (M)
TSPC_GAP_21_6	True	Termination Procedure (M)
TSPC_GAP_21_7	True	LE Ping Procedure (O)
TSPC_GAP_21_8	True	Peripheral Initiated Feature Exchange Procedure (C.1)
TSPC_GAP_21_9	True	Connection Parameter Request Procedure (O)
TSPC_GAP_21_10	True	Data Length Update Procedure (O)
TSPC_GAP_21_11	True	PHY Update Procedure (C.2)
TSPC_GAP_21_12	False	Minimum Number Of Used Channels Procedure (C.2)

Peripheral Discovery Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_22_1	True	Non-Discoverable Mode (M)
TSPC_GAP_22_2	True	Limited Discoverable Mode (O)
TSPC_GAP_22_3	True	General Discoverable Mode (C.1)
TSPC_GAP_22_4	True	Name Discovery Procedure (O)

Peripheral Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_23_1	True	Non-Connectable Mode (M)
TSPC_GAP_23_2	False	Directed Connectable Mode (O)
TSPC_GAP_23_3	True	Undirected Connectable Mode (M)
TSPC_GAP_23_4	True	Connection Parameter Update Procedure (O)
TSPC_GAP_23_5	True	Terminate Connection Procedure (M)
TSPC_GAP_23_6	False	Connected Isochronous Stream Request procedure (C.1)
TSPC_GAP_23_7	False	Connected Isochronous Stream Termination procedure (C.1)

Peripheral Bonding Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_24_1	True	Non-Bondable Mode (M)
TSPC_GAP_24_2	True	Bondable Mode (O)
TSPC_GAP_24_3	True	Bonding Procedure (O)
TSPC_GAP_24_4	True	Multiple Bonds (C.1)

Peripheral Security Aspects Features

Parameter Name	Selected	Description
TSPC_GAP_25_1	True	Security Mode 1 (O)
TSPC_GAP_25_2	True	Security Mode 2 (O)
TSPC_GAP_25_3	True	Authentication Procedure (O)
TSPC_GAP_25_4	True	Authorization Procedure (O)
TSPC_GAP_25_5	True	Connection Data Signing Procedure (O)
TSPC_GAP_25_6	True	Authenticate Signed Data Procedure (O)
TSPC_GAP_25_7	True	Authenticated Pairing (LE security mode 1 level 3) (C.1)
TSPC_GAP_25_8	True	Unauthenticated Pairing (LE security mode 1 level 2) (C.1)
TSPC_GAP_25_9	True	LE Security Mode 1 Level 4 (C.3)
TSPC_GAP_25_10	True	Secure Connections Only Mode (C.4)
TSPC_GAP_25_11	False	Unauthenticated Pairing (LE security mode 1 level 2) with LE Secure Connections Pairing only (C.3)
TSPC_GAP_25_12	False	Authenticated Pairing (LE security mode 1 level 3) with LE Secure Connections Pairing only (C.3)
TSPC_GAP_25_13	True	Minimum 128 Bit entropy key (C.5)

Peripheral Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_26_1	True	Privacy Feature (O)
TSPC_GAP_26_2	True	Non-Resolvable Private Address Generation Procedure (O)
TSPC_GAP_26_3	True	Resolvable Private Address Generation Procedure (C.1)
TSPC_GAP_26_4	True	Resolvable Private Address Resolution Procedure (C.1)

Peripheral GAP Characteristics

Parameter Name	Selected	Description
TSPC_GAP_27_1	True	Device Name (M)
TSPC_GAP_27_2	True	Appearance (M)
TSPC_GAP_27_5	True	Peripheral Preferred Connection Parameters (O)
TSPC_GAP_27_6	True	Writeable Device Name (O)
TSPC_GAP_27_7	False	Writeable Appearance (O)
TSPC_GAP_27_9	True	Central Address Resolution (C.1)

Periodic Advertising Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_27_1	False	Periodic Advertising Synchronization Transfer procedure (C.1)
TSPC_GAP_27_2	False	Periodic Advertising Synchronization Establishment procedure over an LE connection without listening for periodic advertising (C.2)
TSPC_GAP_27_3	False	Periodic Advertising Synchronization Establishment procedure over an LE connection with listening for periodic advertising (C.3)

Central Physical Layer

Parameter Name	Selected	Description
TSPC_GAP_28_1	True	Transmitter (M)
TSPC_GAP_28_2	True	Receiver (M)

Central Link Layer States

Parameter Name	Selected	Description
TSPC_GAP_29_1	True	Standby (M)
TSPC_GAP_29_2	True	Scanning (M)
TSPC_GAP_29_3	True	Initiating (M)
TSPC_GAP_29_4	True	Connection, Central Role (M)

Central Link Layer Scanning Types

Parameter Name	Selected	Description
TSPC_GAP_30_1	True	Passive Scanning (O)
TSPC_GAP_30_2	True	Active Scanning (C.1)

Central Link Layer Control Procedures

Parameter Name	Selected	Description
TSPC_GAP_31_1	True	Connection Update Procedure (M)
TSPC_GAP_31_2	True	Channel Map Update Procedure (M)
TSPC_GAP_31_3	True	Encryption Procedure (O)
TSPC_GAP_31_4	True	Central Initiated Feature Exchange Procedure (M)
TSPC_GAP_31_5	True	Version Exchange Procedure (M)
TSPC_GAP_31_6	True	Termination Procedure (M)
TSPC_GAP_31_7	False	LE Ping Procedure (O)
TSPC_GAP_31_8	True	Peripheral Initiated Feature Exchange Procedure (C.1)
TSPC_GAP_31_9	True	Connection Parameter Request Procedure (O)
TSPC_GAP_31_10	True	Data Length Update Procedure (O)
TSPC_GAP_31_11	True	PHY Update Procedure (C.2)
TSPC_GAP_31_12	False	Minimum Number Of Used Channels Procedure (C.2)

Central Discovery Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_32_1	True	Limited Discovery Procedure (O)
TSPC_GAP_32_2	True	General Discovery Procedure (M)
TSPC_GAP_32_3	True	Name Discovery Procedure (O)

Central Connection Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_33_1	True	Auto Connection Establishment Procedure (O)
TSPC_GAP_33_2	True	General Connection Establishment Procedure (O)
TSPC_GAP_33_3	False	Selective Connection Establishment Procedure (O)
TSPC_GAP_33_4	True	Selective Connection Establishment Procedure (M)
TSPC_GAP_33_5	True	Connection Parameter Update Procedure (M)
TSPC_GAP_33_6	True	Terminate Connection Procedure (M)
TSPC_GAP_33_7	False	Connected Isochronous Stream Creation procedure (C.1)
TSPC_GAP_33_8	False	Connected Isochronous Stream Termination procedure (C.1)

Central Bonding Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_34_1	True	Non-Bondable Mode (M)
TSPC_GAP_34_2	True	Bondable Mode (O)
TSPC_GAP_34_3	True	Bonding Procedure (O)

Central Security Features

Parameter Name	Selected	Description
TSPC_GAP_35_1	True	Security Mode 1 (O)
TSPC_GAP_35_2	True	Security Mode 2 (O)
TSPC_GAP_35_3	True	Authentication Procedure (O)
TSPC_GAP_35_4	False	Authorization Procedure (O)
TSPC_GAP_35_5	True	Connection Data Signing Procedure (O)
TSPC_GAP_35_6	True	Authenticate Signed Data Procedure (O)
TSPC_GAP_35_7	True	Authenticated Pairing (LE security mode 1 level 3) (C.1)
TSPC_GAP_35_8	True	Unauthenticated Pairing (LE security mode1 level 2) (C.1)
TSPC_GAP_35_9	True	LE Security Mode 1 Level 4 (C.2)
TSPC_GAP_35_10	True	Secure Connections Only Mode (C.3)
TSPC_GAP_35_11	False	Unauthenticated Pairing (LE security mode 1 level 2) with LE Secure Connections Pairing only (C.2)
TSPC_GAP_35_12	False	Authenticated Pairing (LE security mode 1 level 3) with LE Secure Connections Pairing only (C.2)
TSPC_GAP_35_13	True	Minimum 128 Bit entropy key (C.4)

Central Privacy Feature

Parameter Name	Selected	Description
TSPC_GAP_36_1	True	Privacy Feature (O)
TSPC_GAP_36_2	True	Non-Resolvable Private Address Generation Procedure (O)
TSPC_GAP_36_3	True	Resolvable Private Address Resolution Procedure (C.1)
TSPC_GAP_36_5	True	Resolvable Private Address Generation Procedure (C.1)

Central GAP Characteristics

Parameter Name	Selected	Description
TSPC_GAP_37_1	True	Device Name (M)
TSPC_GAP_37_2	True	Appearance (M)
TSPC_GAP_37_3	True	Central Address Resolution (C.1)

Periodic Advertising Modes and Procedures

Parameter Name	Selected	Description
TSPC_GAP_37_1	False	Periodic Advertising Synchronization Transfer procedure (C.1)
TSPC_GAP_37_2	False	Periodic Advertising Synchronization Establishment procedure over an LE connection without listening for periodic advertising (C.2)
TSPC_GAP_37_3	False	Periodic Advertising Synchronization Establishment procedure over an LE connection with listening for periodic advertising (C.3)

BR/EDR/LE Roles

Parameter Name	Selected	Description
TSPC_GAP_38_1	False	Broadcaster (C.1)
TSPC_GAP_38_2	False	Observer (C.1)
TSPC_GAP_38_3	False	Peripheral (C.1)
TSPC_GAP_38_4	False	Central (C.1)

Central BR/EDR/LE Security Aspects

Parameter Name	Selected	Description
TSPC_GAP_41_1	False	Security Aspects (M)
TSPC_GAP_41_2a	False	Derivation of BR/EDR Link Key from LE LTK (C.1)
TSPC_GAP_41_2b	False	Derivation of LE LTK from BR/EDR Link Key (C.2)

Peripheral BR/EDR/LE Security Aspects

Parameter Name	Selected	Description
TSPC_GAP_43_1	False	Security Aspects (M)
TSPC_GAP_43_2a	False	Derivation of BR/EDR Link Key from LE LTK (C.1)
TSPC_GAP_43_2b	False	Derivation of LE LTK from BR/EDR Link Key (C.2)

Central Simultaneous BR/EDR and LE Transports

Parameter Name	Selected	Description
TSPC_GAP_44_1	False	Simultaneous BR/EDR and LE Transports – BR/EDR Peripheral to the same device (O)
TSPC_GAP_44_2	False	Simultaneous BR/EDR and LE Transports – BR/EDR Central to the same device (O)

Peripheral Simultaneous BR/EDR and LE Transports

Parameter Name	Selected	Description
TSPC_GAP_45_1	False	Simultaneous BR/EDR and LE Transports – BR/EDR Peripheral to the same device (O)
TSPC_GAP_45_2	False	Simultaneous BR/EDR and LE Transports – BR/EDR Central to the same device (O)

GATT ICS PTS version: 8.0.3

M - mandatory

O - optional

Generic Attribute Profile Support

Parameter Name	Selected	Description
TSPC_GATT_1_1	True	Generic Attribute Profile (GATT) Client (C.1)
TSPC_GATT_1_2	True	Generic Attribute Profile (GATT) Server (C.2)

GATT role configuration

Parameter Name	Selected	Description
TSPC_GATT_1a_1	True	GATT Client over LE (C.1)
TSPC_GATT_1a_2	False	GATT Client over BR/EDR (C.2)
TSPC_GATT_1a_3	True	GATT Server over LE (C.3)
TSPC_GATT_1a_4	False	GATT Server over BR/EDR (C.4)

Attribute Protocol Transport

Parameter Name	Selected	Description
TSPC_GATT_2_1	False	Attribute Protocol Supported over BR/EDR (L2CAP fixed channel support) (C.1)
TSPC_GATT_2_2	True	Attribute Protocol Supported over LE (C.2)
TSPC_GATT_2_3	True	Enhanced ATT bearer Attribute Protocol Supported (L2CAP fixed EATT PSM supported) (C.3)
TSPC_GATT_2_3a	True	Enhanced ATT bearer supported over LE (C.4)
TSPC_GATT_2_3b	False	Enhanced ATT bearer supported over BR/EDR (C.5)

Generic Attribute Profile Feature Support, by Client

Parameter Name	Selected	Description
TSPC_GATT_3_1	True	Exchange MTU (C.11)
TSPC_GATT_3_2	True	Discover All Primary Services (O)
TSPC_GATT_3_3	True	Discover Primary Services by Service UUID (O)
TSPC_GATT_3_4	True	Find Included Services (O)
TSPC_GATT_3_5	True	Discover All characteristics of a Service (O)
TSPC_GATT_3_6	True	Discover Characteristics by UUID (O)
TSPC_GATT_3_7	True	Discover All Characteristic Descriptors (O)
TSPC_GATT_3_8	True	Read Characteristic Value (O)
TSPC_GATT_3_9	True	Read Using Characteristic UUID (O)
TSPC_GATT_3_10	True	Read Long Characteristic Values (O)
TSPC_GATT_3_11	True	Read Multiple Characteristic Values (O)
TSPC_GATT_3_12	True	Write without Response (O)
TSPC_GATT_3_13	True	Signed Write Without Response (C.11)
TSPC_GATT_3_14	True	Write Characteristic Value (O)
TSPC_GATT_3_15	True	Write Long Characteristic Values (O)
TSPC_GATT_3_16	True	Characteristic Value Reliable Writes (O)
TSPC_GATT_3_17	True	Notifications (C.7)
TSPC_GATT_3_18	True	Indications (M)
TSPC_GATT_3_19	True	Read Characteristic Descriptors (O)
TSPC_GATT_3_20	True	Read Long Characteristic Descriptors (O)
TSPC_GATT_3_21	True	Write Characteristic Descriptors (O)
TSPC_GATT_3_22	True	Write Long Characteristic Descriptors (O)
TSPC_GATT_3_23	True	Service Changed Characteristic (M)
TSPC_GATT_3_24	False	Configured Broadcast (C.2)
TSPC_GATT_3_25	True	Client Supported Features Characteristic (C.4)

continues on next page

Table 1 – continued from previous page

Parameter Name	Selected	Description
TSPC_GATT_3_26	True	Database Hash Characteristic (C.4)
TSPC_GATT_3_27	False	Read and Interpret Characteristic Presentation Format (O)
TSPC_GATT_3_28	False	Read and Interpret Characteristic Aggregate Format (C.6)
TSPC_GATT_3_29	False	Read Multiple Variable Length Characteristic Values (C.9)
TSPC_GATT_3_30	False	Multiple Variable Length Notifications (C.10)

Generic Attribute Profile Feature Support, by Server

Parameter Name	Selected	Description
TSPC_GATT_4_1	True	Exchange MTU (C.6)
TSPC_GATT_4_2	True	Discover All Primary Services (M)
TSPC_GATT_4_3	True	Discover Primary Services by Service UUID (M)
TSPC_GATT_4_4	True	Find Included Services (M)
TSPC_GATT_4_5	True	Discover All characteristics of a Service (M)
TSPC_GATT_4_6	True	Discover Characteristics by UUID (M)
TSPC_GATT_4_7	True	Discover All Characteristic Descriptors (M)
TSPC_GATT_4_8	True	Read Characteristic Value (M)
TSPC_GATT_4_9	True	Read Using Characteristic UUID (M)
TSPC_GATT_4_10	True	Read Long Characteristic Values (C.12)
TSPC_GATT_4_11	True	Read Multiple Characteristic Values (O)
TSPC_GATT_4_12	True	Write without Response (C.2)
TSPC_GATT_4_13	True	Signed Write Without Response (C.6)
TSPC_GATT_4_14	True	Write Characteristic Value (C.3)
TSPC_GATT_4_15	True	Write Long Characteristic Values (C.12)
TSPC_GATT_4_16	True	Characteristic Value ReliableWrites (O)
TSPC_GATT_4_17	True	Notifications (O)
TSPC_GATT_4_18	True	Indications (C.1)
TSPC_GATT_4_19	True	Read Characteristic Descriptors (C.12)
TSPC_GATT_4_20	True	Read Long Characteristic Descriptors (C.12)
TSPC_GATT_4_21	True	Write Characteristic Descriptors (C.12)
TSPC_GATT_4_22	True	Write Long Characteristic Descriptors (O)
TSPC_GATT_4_23	True	Service Changed Characteristic (C.1)
TSPC_GATT_4_24	False	Configured Broadcast (C.5)
TSPC_GATT_4_25	False	Execute Write Request with empty queue (C.7)
TSPC_GATT_4_26	True	Client Supported Features Characteristic (C.9)
TSPC_GATT_4_27	True	Database Hash Characteristic (C.8)
TSPC_GATT_4_28	False	Report Characteristic Value: Characteristic Presentation Format (O)
TSPC_GATT_4_29	False	Report aggregate Characteristic Value: Characteristic Aggregate Format (C.10)
TSPC_GATT_4_30	False	Read Multiple Variable Length Characteristic Values (C.13)
TSPC_GATT_4_31	False	Multiple Variable Length Notifications (C.13)

SDP Interoperability

Parameter Name	Selected	Description
TSPC_GATT_6_2	False	Discover GATT Services using Service Discovery Profile (C.1)
TSPC_GATT_6_3	False	Publish SDP record for GATT services support via BR/EDR (C.2)

Attribute Protocol Transport Security

Parameter Name	Selected	Description
TSPC_GATT_7_1	False	Security Mode 4 (C.1)
TSPC_GATT_7_2	True	LE Security Mode 1 (C.5)
TSPC_GATT_7_3	True	LE Security Mode 2 (C.6)
TSPC_GATT_7_4	True	LE Authentication Procedure (C.4)
TSPC_GATT_7_5	True	LE connection data signing procedure (C.2)
TSPC_GATT_7_6	True	LE Authenticate signed data procedure (C.2)
TSPC_GATT_7_7	True	LE Authorization Procedure (C.3)

Multiple Simultaneous ATT Bearers

Parameter Name	Selected	Description
TSPC_GATT_8_1	False	Support for multiple simultaneous active ATT bearers from same device – ATT over LE and ATT over BR/EDR (C.1)
TSPC_GATT_8_2	True	Support for multiple simultaneous active ATT bearers from same device – ATT over LE and EATT over LE (C.2)
TSPC_GATT_8_3	False	Support for multiple simultaneous active ATT bearers from same device – ATT over BR/EDR and EATT over BR/EDR (C.3)
TSPC_GATT_8_4	False	Support for multiple simultaneous active ATT bearers from same device – ATT over LE and EATT over BR/EDR (C.4)
TSPC_GATT_8_5	False	Support for multiple simultaneous active ATT bearers from same device – ATT over BR/EDR and EATT over LE (C.5)
TSPC_GATT_8_6	False	Support for multiple simultaneous active EATT bearers from same device – EATT over BR/EDR and EATT over LE (C.6)
TSPC_GATT_8_7	False	Support for multiple simultaneous active EATT bearers from same device – EATT over BR/EDR (C.7)
TSPC_GATT_8_8	True	Support for multiple simultaneous active EATT bearers from same device – EATT over LE (C.7)

L2CAP ICS PTS version: 8.0.3

M - mandatory

O - optional

L2CAP Transport Configuration

Parameter Name	Selected	Description
TSPC_L2CAP_0_1	False	BR/EDR (includes possible support of GAP LE Broadcaster or LE Observer roles) (C.1)
TSPC_L2CAP_0_2	True	LE (C.2)
TSPC_L2CAP_0_3	False	BR/EDR/LE (C.3)

Roles

Parameter Name	Selected	Description
TSPC_L2CAP_1_1	False	Data Channel Initiator (C.3)
TSPC_L2CAP_1_2	False	Data Channel Acceptor (C.1)
TSPC_L2CAP_1_3	True	LE Master (C.2)
TSPC_L2CAP_1_4	True	LE Slave (C.2)
TSPC_L2CAP_1_5	True	LE Data Channel Initiator (C.4)
TSPC_L2CAP_1_6	True	LE Data Channel Acceptor (C.5)

General Operation

Parameter Name	Selected	Description
TSPC_L2CAP_2_1	False	Support of L2CAP signalling channel (C.16)
TSPC_L2CAP_2_2	False	Support of configuration process (C.16)
TSPC_L2CAP_2_3	False	Support of connection oriented data channel (C.16)
TSPC_L2CAP_2_4	False	Support of command echo request (C.17)
TSPC_L2CAP_2_5	False	Support of command echo response (C.16)
TSPC_L2CAP_2_6	False	Support of command information request (C.17)
TSPC_L2CAP_2_7	False	Support of command information response (C.16)
TSPC_L2CAP_2_8	False	Support of a channel group (C.17)
TSPC_L2CAP_2_9	False	Support of packet for connectionless channel (C.17)
TSPC_L2CAP_2_10	False	Support retransmission mode (C.17)
TSPC_L2CAP_2_11	False	Support flow control mode (C.17)
TSPC_L2CAP_2_12	False	Enhanced Retransmission Mode (C.11)
TSPC_L2CAP_2_13	False	Streaming Mode (O)
TSPC_L2CAP_2_14	False	FCS Option (C.1)
TSPC_L2CAP_2_15	False	Generate Local Busy Condition (C.2)
TSPC_L2CAP_2_16	False	Send Reject (C.2)
TSPC_L2CAP_2_17	False	Send Selective Reject (C.2)
TSPC_L2CAP_2_18	False	Mandatory use of ERTM (C.3)
TSPC_L2CAP_2_19	False	Mandatory use of Streaming Mode (C.4)
TSPC_L2CAP_2_20	False	Optional use of ERTM (C.3)
TSPC_L2CAP_2_21	False	Optional use of Streaming Mode (C.4)
TSPC_L2CAP_2_22	False	Send data using SAR in ERTM (C.5)
TSPC_L2CAP_2_23	False	Send data using SAR in Streaming Mode (C.6)
TSPC_L2CAP_2_24	False	Actively request Basic Mode for a PSM that supports the use of ERTM or Streaming M
TSPC_L2CAP_2_25	False	Supports performing L2CAP channel mode configuration fallback from SM to ERTM (
TSPC_L2CAP_2_26	False	Supports sending more than one unacknowledged I-Frame when operating in ERTM
TSPC_L2CAP_2_27	False	Supports sending more than three unacknowledged I-Frame when operating in ERTM
TSPC_L2CAP_2_28	False	Supports configuring the peer TxWindow greater than 1. (C.10)
TSPC_L2CAP_2_29	False	AMP Support (C.11)
TSPC_L2CAP_2_30	False	Fixed Channel Support (C.11)
TSPC_L2CAP_2_31	False	AMP Manager Support (C.11)
TSPC_L2CAP_2_32	False	ERTM over AMP (C.11)
TSPC_L2CAP_2_33	False	Streaming Mode Source over AMP Support (C.12)
TSPC_L2CAP_2_34	False	Streaming Mode Sink over AMP Support (C.12)
TSPC_L2CAP_2_35	False	Unicast Connectionless Data, Reception (O)
TSPC_L2CAP_2_36	False	Ability to transmit an unencrypted packet over a unicast connectionless L2CAP chann
TSPC_L2CAP_2_37	False	Ability to transmit an encrypted packet over a unicast connectionless L2CAP channel.
TSPC_L2CAP_2_38	False	Extended Flow Specification for BR/EDR (C.7)
TSPC_L2CAP_2_39	False	Extended Window Size (C.7)
TSPC_L2CAP_2_40	True	Support of Low Energy signaling channel (C.13)
TSPC_L2CAP_2_41	True	Support of command reject (C.13)
TSPC_L2CAP_2_42	True	Send Connection Parameter Update Request (C.14)
TSPC_L2CAP_2_43	True	Send Connection Parameter Update Response (C.15)
TSPC_L2CAP_2_44	False	Extended Flow Specification for AMP (C.18)
TSPC_L2CAP_2_45	False	Send Disconnect Request Command (C.21)
TSPC_L2CAP_2_45a	True	Send Disconnect Request Command – LE (C.22)
TSPC_L2CAP_2_46	True	Support LE Credit Based Flow Control Mode (C.19)
TSPC_L2CAP_2_47	True	Support for LE Data Channel (C.20)
TSPC_L2CAP_2_48	True	Support Enhanced Credit Based Flow Control Mode (C.23)

Configurable Parameters

Parameter Name	Selected	Description
TSPC_L2CAP_3_1	True	Support of RTX timer (M)
TSPC_L2CAP_3_2	False	Support of ERTX timer (C.4)
TSPC_L2CAP_3_3	False	Support minimum MTU size 48 octets (C.4)
TSPC_L2CAP_3_4	False	Support MTU size larger than 48 octets (C.5)
TSPC_L2CAP_3_5	False	Support of flush timeout value for reliable channel (C.4)
TSPC_L2CAP_3_6	False	Support of flush timeout value for unreliable channel (C.5)
TSPC_L2CAP_3_7	False	Support of bi-directional quality of service (QoS) option field (C.1)
TSPC_L2CAP_3_8	False	Negotiate QoS service type (C.5)
TSPC_L2CAP_3_9	False	Negotiate and support service type 'No Traffic' (C.2)
TSPC_L2CAP_3_10	False	Negotiate and support service type 'Best effort' (C.3)
TSPC_L2CAP_3_11	False	Negotiate and support service type 'Gauranteed' (C.2)
TSPC_L2CAP_3_12	True	Support minimum MTU size 23 octets (C.6)
TSPC_L2CAP_3_13	False	Negotiate and support service type 'No traffic' for Extended Flow Specification (C.7)
TSPC_L2CAP_3_14	False	Negotiate and support service type 'Best Effort' for Extended Flow Specification (C.8)
TSPC_L2CAP_3_15	False	Negotiate and support service type 'Guaranteed' for Extended Flow Specification. (C.9)
TSPC_L2CAP_3_16	True	Support Multiple Simultaneous LE Data Channels (C.10)

SM ICS PTS version: 8.0.3

M - mandatory

O - optional

Role

Parameter Name	Selected	Description
TSPC_SM_1_1	True	Central Role (Initiator) (C.1)
TSPC_SM_1_2	True	Peripheral Role (Responder) (C.2)

Security Properties

Parameter Name	Selected	Description
TSPC_SM_2_1	True	Authenticated MITM protection (O)
TSPC_SM_2_2	True	Unauthenticated no MITM protection (C.1)
TSPC_SM_2_3	True	No security requirements (M)
TSPC_SM_2_4	True	OOB supported (O)
TSPC_SM_2_5	True	LE Secure Connections (O)

Encryption Key Size

Parameter Name	Selected	Description
TSPC_SM_3_1	True	Encryption Key Size (M)

Pairing Method

Parameter Name	Selected	Description
TSPC_SM_4_1	True	Just Works (O)
TSPC_SM_4_2	True	Passkey Entry (C.1)
TSPC_SM_4_3	True	Out of Band (C.1)

Security Initiation

Parameter Name	Selected	Description
TSPC_SM_5_1	True	Encryption Setup using STK (C.3)
TSPC_SM_5_2	True	Encryption Setup using LTK (O)
TSPC_SM_5_3	True	Peripheral Initiated Security (C.1)
TSPC_SM_5_4	True	Peripheral Initiated Security – Central response (C.2)
TSPC_SM_5_5	False	Link Key Conversion Function h7 (C.4)
TSPC_SM_5_6	False	Link Key Conversion Function h6 (C.5)

Signing Algorithm

Parameter Name	Selected	Description
TSPC_SM_6_1	True	Signing Algorithm - Generation (O)
TSPC_SM_6_2	True	Signing Algorithm - Resolving (O)

Key Distribution

Parameter Name	Selected	Description
TSPC_SM_7_1	True	Encryption Key (C.1)
TSPC_SM_7_2	True	Identity Key (C.2)
TSPC_SM_7_3	True	Signing Key (C.3)

Cross-Transport Key Derivation

Parameter Name	Selected	Description
TSPC_SM_8_1	False	Cross Transport Key Derivation Supported (C.1)
TSPC_SM_8_2	False	Derivation of LE LTK from BR/EDR Link Key (C.2)
TSPC_SM_8_3	False	Derivation of BR/EDR Link Key from LE LTK (C.2)

RFCOMM PICS PTS version: 6.4

- – different than PTS defaults

Protocol Version

Parameter Name	Selected	Description
TSPC_RFCOMM_0_1	False	RFCOMM 1.1 with TS 07.10
TSPC_RFCOMM_0_2	True (*)	RFCOMM 1.2 with TS 07.10

Supported Procedures

Parameter Name	Selected	Description
TSPC_RFCOMM_1_1	True (*)	Initialize RFCOMM Session
TSPC_RFCOMM_1_2	True (*)	Respond to Initialization of an RFCOMM Session
TSPC_RFCOMM_1_3	True	Shutdown RFCOMM Session
TSPC_RFCOMM_1_4	True	Respond to a Shutdown of an RFCOMM Session
TSPC_RFCOMM_1_5	True (*)	Establish DLC
TSPC_RFCOMM_1_6	True (*)	Respond to Establishment of a DLC
TSPC_RFCOMM_1_7	True	Disconnect DLC
TSPC_RFCOMM_1_8	True	Respond to Disconnection of a DLC
TSPC_RFCOMM_1_9	True	Respond to and send MSC Command
TSPC_RFCOMM_1_10	True	Initiate Transfer Information
TSPC_RFCOMM_1_11	True	Respond to Test Command
TSPC_RFCOMM_1_12	False	Send Test Command
TSPC_RFCOMM_1_13	True	React to Aggregate Flow Control
TSPC_RFCOMM_1_14	True	Respond to RLS Command
TSPC_RFCOMM_1_15	False	Send RLS Command
TSPC_RFCOMM_1_16	True	Respond to PN Command
TSPC_RFCOMM_1_17	True (*)	Send PN Command
TSPC_RFCOMM_1_18	True (*)	Send Non-Supported Command (NSC) response
TSPC_RFCOMM_1_19	True	Respond to RPN Command
TSPC_RFCOMM_1_20	False	Send RPN Command
TSPC_RFCOMM_1_21	True (*)	Closing Multiplexer by First Sending a DISC Command
TSPC_RFCOMM_1_22	True	Support of Credit Based Flow Control

MESH ICS PTS version: 8.0.3

M - mandatory

O - optional

Major Profile Version (X.Y)

Parameter Name	Selected	Description
TSPC_MESH_0_1	True	Mesh v1.0 (M)

Minor Profile Version (X.Y.Z)

Parameter Name	Selected	Description
TSPC_MESH_0a_1	True	Erratum 10395 (C.1)
TSPC_MESH_0a_2	True	Mesh v1.0.1 (C.2)
TSPC_MESH_0a_3	True	Erratum 16350 (C.1)

Roles

Parameter Name	Selected	Description
TSPC_MESH_2_1	True	Node (C.1)
TSPC_MESH_2_2	False	Provisioner (C.1)

Node Capabilities - Bearers

Parameter Name	Selected	Description
TSPC_MESH_3_1	True	Advertising Bearer (C.1)
TSPC_MESH_3_2	True	GATT Bearer (C.1)

Node Capabilities - Provisioning

Parameter Name	Selected	Description
TSPC_MESH_4_1	True	PB-ADV (C.1)
TSPC_MESH_4_2	True	PB-GATT (C.2)
TSPC_MESH_4_3	True	Device UUID (C.3)
TSPC_MESH_4_4	True	Sending Unprovisioned Device Beacon (C.4)
TSPC_MESH_4_5	True	Generic Provisioning Layer (C.3)
TSPC_MESH_4_6	True	Provisioning Protocol (Provisioning Server) (C.3)
TSPC_MESH_4_7	False	Provisioning: Public Key OOB (C.5)
TSPC_MESH_4_8	True	Provisioning: Public Key No OOB (C.5)
TSPC_MESH_4_9	True	Provisioning: Authentication Output OOB (C.6)
TSPC_MESH_4_10	False	Provisioning: Authentication Input OOB (C.6)
TSPC_MESH_4_11	False	Provisioning: Authentication Static OOB (C.6)
TSPC_MESH_4_12	True	Provisioning: Authentication No OOB (C.3)

Node Capabilities – Network Layer

Parameter Name	Selected	Description
TSPC_MESH_5_1	True	Transmitting and Receiving Secured Network Layer Messages (M)
TSPC_MESH_5_2	True	Relay Feature (C.1)
TSPC_MESH_5_3	True	Network Message Cache (C.2)
TSPC_MESH_5_4	False	Multiple Subnet Support (O)

Node Capabilities – Lower Transport Layer

Parameter Name	Selected	Description
TSPC_MESH_6_1	True	Transmitting and Receiving a Lower Transport PDU (M)
TSPC_MESH_6_2	True	Segmentation and Reassembly Behavior (M)
TSPC_MESH_6_3	True	Friend Cache (C.1)

Node Capabilities – Upper Transport Layer

Parameter Name	Selected	Description
TSPC_MESH_7_1	True	Transmitting a Secured Access Payload (M)
TSPC_MESH_7_2	True	Receiving a Secured Upper Transport PDU (M)
TSPC_MESH_7_3	True	Friend Feature (C.1)
TSPC_MESH_7_4	True	Low Power Feature (C.1)
TSPC_MESH_7_5	True	Heartbeat (M)

Node Capabilities – Access Layer

Parameter Name	Selected	Description
TSPC_MESH_8_1	True	Transmitting and Receiving an Access Layer Message (M)

Node Capabilities – Security

Parameter Name	Selected	Description
TSPC_MESH_9_1	True	Message Replay Protection (M)

Node Capabilities – Mesh Management

Parameter Name	Selected	Description
TSPC_MESH_10_1	True	Secure Network Beacon (M)
TSPC_MESH_10_2	True	Key Refresh Procedure (M)
TSPC_MESH_10_3	True	IV Update Procedure (M)
TSPC_MESH_10_4	True	IV Index Recovery Procedure (M)

Node Capabilities – Foundation Mesh Models

Parameter Name	Selected	Description
TSPC_MESH_11_1	True	Configuration Server Model (M)
TSPC_MESH_11_2	True	Health Server Model (M)
TSPC_MESH_11_3	False	Configuration Client Model (O)
TSPC_MESH_11_4	False	Health Client Model (O)

Node Capabilities – Proxy

Parameter Name	Selected	Description
TSPC_MESH_12_1	True	Proxy Server (C.1)
TSPC_MESH_12_2	True	GATT Server (C.2)
TSPC_MESH_12_3	True	Advertising with Network ID (C.2)
TSPC_MESH_12_4	True	Advertising with Node Identity (C.2)
TSPC_MESH_12_5	False	Proxy Client (C.3)
TSPC_MESH_12_6	False	GATT Client (C.4)

Mesh GATT Services

Parameter Name	Selected	Description
TSPC_MESH_13_1	True	Mesh Provisioning Service (C.1)
TSPC_MESH_13_2	True	Mesh Proxy Service (C.2)

GATT Server Requirements

Parameter Name	Selected	Description
TSPC_MESH_14_1	True	Discover All Primary Services (M)
TSPC_MESH_14_2	True	Discover Primary Services by Service UUID (M)
TSPC_MESH_14_3	True	Write without Response (M)
TSPC_MESH_14_4	True	Notifications (M)
TSPC_MESH_14_5	True	Write Characteristic Descriptors (M)

GATT Client Requirements

Parameter Name	Selected	Description
TSPC_MESH_15_1	False	Discover All Primary Services (C.1)
TSPC_MESH_15_2	False	Discover Primary Services by Service UUID (C.1)
TSPC_MESH_15_3	False	Write without Response (M)
TSPC_MESH_15_4	False	Notifications (M)
TSPC_MESH_15_5	False	Write Characteristic Descriptors (M)

GAP Requirements

Parameter Name	Selected	Description
TSPC_MESH_16_1	True	Broadcaster (C.1)
TSPC_MESH_16_2	True	Observer (C.1)
TSPC_MESH_16_3	True	Peripheral (C.2)
TSPC_MESH_16_4	True	Peripheral – Security Mode 1 (C.2)
TSPC_MESH_16_5	False	Central (C.3)
TSPC_MESH_16_6	False	Central – Security Mode 1 (C.3)

Provisioner – Bearers

Parameter Name	Selected	Description
TSPC_MESH_17_1	False	Advertising Bearer (C.1)
TSPC_MESH_17_2	False	GATT Bearer (C.1)

Provisioner – Provisioning

Parameter Name	Selected	Description
TSPC_MESH_18_1	False	Receiving Unprovisioned Device Beacon (C.1)
TSPC_MESH_18_2	False	PB-ADV (C.1)
TSPC_MESH_18_3	False	Generic Provisioning Layer (M)
TSPC_MESH_18_4	False	Provisioning Protocol (Provisioning Client) (M)
TSPC_MESH_18_5	False	PB-GATT (C.2)
TSPC_MESH_18_6	False	GATT Client (C.2)
TSPC_MESH_18_7	False	Provisioning: Public Key OOB (M)
TSPC_MESH_18_8	False	Provisioning: Public Key No OOB (M)
TSPC_MESH_18_9	False	Provisioning: Authentication Output OOB (M)
TSPC_MESH_18_10	False	Provisioning: Authentication Input OOB (M)
TSPC_MESH_18_11	False	Provisioning: Authentication Static or No OOB (M)

Provisioner – Mesh Management

Parameter Name	Selected	Description
TSPC_MESH_19_1	False	Receiving Secure Network Beacon (M)

GATT Client Requirements

Parameter Name	Selected	Description
TSPC_MESH_20_1	False	Discover All Primary Services (C.1)
TSPC_MESH_20_2	False	Discover Primary Services by Service UUID (C.1)
TSPC_MESH_20_3	False	Write without Response (M)
TSPC_MESH_20_4	False	Notifications (M)
TSPC_MESH_20_5	False	Write Characteristic Descriptors (M)

GAP Requirements

Parameter Name	Selected	Description
TSPC_MESH_21_1	False	Broadcaster (C.1)
TSPC_MESH_21_2	False	Observer (C.1)
TSPC_MESH_21_3	False	Central (C.2)
TSPC_MESH_21_4	False	Central - Security Mode 1 (C.2)

DIS ICS PTS version: 8.0.3

M - mandatory

O - optional

Service Version

Parameter Name	Selected	Description
TSPC_DIS_0_1	True	Device Information Service v1.1 (M)

Transport Requirements

Parameter Name	Selected	Description
TSPC_DIS_1_1	False	Service supported over BR/EDR (C.1)
TSPC_DIS_1_2	True	Service supported over LE (C.1)
TSPC_DIS_1_3	False	Service supported over HS (C.1)

Service Requirements

Parameter Name	Selected	Description
TSPC_DIS_2_1	True	Device Information Service (M)
TSPC_DIS_2_2	True	Manufacturer Name String Characteristic (O)
TSPC_DIS_2_3	True	Model Number String Characteristic (O)
TSPC_DIS_2_4	True	Serial Number String Characteristic (O)
TSPC_DIS_2_5	True	Hardware Revision String Characteristic (O)
TSPC_DIS_2_6	True	Firmware Revision String Characteristic (O)
TSPC_DIS_2_7	True	Software Revision String Characteristic (O)
TSPC_DIS_2_8	False	System ID Characteristic (O)
TSPC_DIS_2_9	False	IEEE 11073-20601 Regulatory Certification Data List Characteristic (O)
TSPC_DIS_2_10	False	SDP Interoperability (C.1)
TSPC_DIS_2_11	True	PnP ID (O)

8.3.4 Bluetooth tools

This page lists and describes tools that can be used to assist during Bluetooth stack or application development in order to help, simplify and speed up the development process.

Mobile applications

It is often useful to make use of existing mobile applications to interact with hardware running Zephyr, to test functionality without having to write any additional code or requiring extra hardware.

The recommended mobile applications for interacting with Zephyr are:

- Android:
  - nRF Connect for Android
  - nRF Mesh for Android
  - LightBlue for Android
- iOS:
  - nRF Connect for iOS
  - nRF Mesh for iOS
  - LightBlue for iOS

Using BlueZ with Zephyr

The Linux Bluetooth Protocol Stack, BlueZ, comes with a very useful set of tools that can be used to debug and interact with Zephyr's BLE Host and Controller. In order to benefit from these tools you will need to make sure that you are running a recent version of the Linux Kernel and BlueZ:

- Linux Kernel 4.10+
- BlueZ 4.45+

Additionally, some of the BlueZ tools might not be bundled by default by your Linux distribution. If you need to build BlueZ from scratch to update to a recent version or to obtain all of its tools you can follow the steps below:

```
git clone git://git.kernel.org/pub/scm/bluetooth/bluez.git
cd bluez
./bootstrap-configure --disable-android --disable-midi
make
```

You can then find `btattach`, `btmgt` and `btproxy` in the `tools/` folder and `btmon` in the `monitor/` folder.

You'll need to enable BlueZ's experimental features so you can access its most recent BLE functionality. Do this by editing the file `/lib/systemd/system/bluetooth.service` and making sure to include the `-E` option in the daemon's execution start line:

```
ExecStart=/usr/libexec/bluetooth/bluetoothd -E
```

Finally, reload and restart the daemon:

```
sudo systemctl daemon-reload
sudo systemctl restart bluetooth
```

Running on QEMU and Native POSIX

It's possible to run Bluetooth applications using either the *QEMU emulator* or Native POSIX. In either case, a Bluetooth controller needs to be exported from the host OS (Linux) to the emulator. For this purpose you will need some tools described in the *Using BlueZ with Zephyr* section.

Using the Host System Bluetooth Controller The host OS's Bluetooth controller is connected in the following manner:

- To the second QEMU serial line using a UNIX socket. This socket gets used with the help of the QEMU option `-serial unix:/tmp/bt-server-bredr`. This option gets passed to QEMU through `QEMU_EXTRA_FLAGS` automatically whenever an application has enabled Bluetooth support.
- To a serial port in Native POSIX through the use of a command-line option passed to the Native POSIX executable: `--bt-dev=hci0`

On the host side, BlueZ allows you to export its Bluetooth controller through a so-called user channel for QEMU and Native POSIX to use.

Note: You only need to run `btproxy` when using QEMU. Native POSIX handles the UNIX socket proxying automatically

---

If you are using QEMU, in order to make the Controller available you will need one additional step using `btproxy`:

1. Make sure that the Bluetooth controller is down
2. Use the btproxy tool to open the listening UNIX socket, type:

```
sudo tools/btproxy -u -i 0
Listening on /tmp/bt-server-bredr
```

You might need to replace `-i 0` with the index of the Controller you wish to proxy.

Once the hardware is connected and ready to use, you can then proceed to building and running a sample:

- Choose one of the Bluetooth sample applications located in `samples/bluetooth`.
- To run a Bluetooth application in QEMU, type:

```
west build -b qemu_x86 samples/bluetooth/<sample>
west build -t run
```

Running QEMU now results in a connection with the second serial line to the `bt-server-bredr` UNIX socket, letting the application access the Bluetooth controller.

- To run a Bluetooth application in Native POSIX, first build it:

```
west build -b native_posix samples/bluetooth/<sample>
```

And then run it with:

```
$ sudo ./build/zephyr/zephyr.exe --bt-dev=hci0
```

Using a Zephyr-based BLE Controller Depending on which hardware you have available, you can choose between two transports when building a single-mode, Zephyr-based BLE Controller:

- UART: Use the hci_uart sample and follow the instructions in bluetooth-hci-uart-qemu-posix.
- USB: Use the hci_usb sample and then treat it as a Host System Bluetooth Controller (see previous section)

HCI Tracing When running the Host on a computer connected to an external Controller, it is very useful to be able to see the full log of exchanges between the two, in the format of a *Host Controller Interface* log. In order to see those logs, you can use the built-in `btmon` tool from BlueZ:

```
$ btmon
```

Using Zephyr-based Controllers with BlueZ

If you want to test a Zephyr-powered BLE Controller using BlueZ's Bluetooth Host, you will need a few tools described in the *Using BlueZ with Zephyr* section. Once you have installed the tools you can then use them to interact with your Zephyr-based controller:

```
sudo tools/btmgmt --index 0
[hci0]# auto-power
[hci0]# find -l
```

You might need to replace `--index 0` with the index of the Controller you wish to manage. Additional information about `btmgmt` can be found in its manual pages.

8.3.5 Developing Bluetooth Applications

Bluetooth applications are developed using the common infrastructure and approach that is described in the *Application Development* section of the documentation.

Additional information that is only relevant to Bluetooth applications can be found in this page.

Hardware setup

This section describes the options you have when building and debugging Bluetooth applications with Zephyr. Depending on the hardware that is available to you, the requirements you have and the type of development you prefer you may pick one or another setup to match your needs.

There are 4 possible hardware setups to use with Zephyr and Bluetooth:

1. Embedded
2. QEMU with an external Controller
3. Native POSIX with an external Controller
4. Simulated nRF52 with BabbleSim

Embedded This setup relies on all software running directly on the embedded platform(s) that the application is targeting. All the *Configurations* and *Build Types* are supported but you might need to build Zephyr more than once if you are using a dual-chip configuration or if you have multiple cores in your SoC each running a different build type (e.g., one running the Host, the other the Controller).

To start developing using this setup follow the *Getting Started Guide*, choose one (or more if you are using a dual-chip solution) boards that support Bluetooth and then *run the application*).

Embedded HCI tracing When running both Host and Controller in actual Integrated Circuits, you will only see normal log messages on the console by default, without any way of accessing the HCI traffic between the Host and the Controller. However, there is a special Bluetooth logging mode that converts the console to use a binary protocol that interleaves both normal log messages as well as the HCI traffic. Set the following Kconfig options to enable this protocol before building your application:

```
CONFIG_BT_DEBUG_MONITOR_UART=y
CONFIG_UART_CONSOLE=n
```

Setting `CONFIG_BT_DEBUG_MONITOR_UART` to `y` replaces the `CONFIG_BT_DEBUG_LOG` option, and setting `CONFIG_UART_CONSOLE` to `n` disables the default `printk/printf` hooks.

To decode the binary protocol that will now be sent to the console UART you need to use the btmon tool from *BlueZ*:

```
$ btmon --tty <console TTY> --tty-speed 115200
```

Host on Linux with an external Controller

Note: This is currently only available on GNU/Linux

This setup relies on a "dual-chip" *configuration* which is comprised of the following devices:

1. A *Host-only* application running in the *QEMU* emulator or the `native_posix` native port of Zephyr
2. A Controller, which can be one of two types:
   - A commercially available Controller
   - A *Controller-only* build of Zephyr

Warning: Certain external Controllers are either unable to accept the Host to Controller flow control parameters that Zephyr sets by default (Qualcomm), or do not transmit any data from the Controller to the Host (Realtek). If you see a message similar to:

```
<wrn> bt_hci_core: opcode 0x0c33 status 0x12
```

when booting your sample of choice (make sure you have enabled `CONFIG_BT_DEBUG_LOG` in your `prj.conf` before running the sample), or if there is no data flowing from the Controller to the Host, then you need to disable Host to Controller flow control. To do so, set `CONFIG_BT_HCI_ACL_FLOW_CONTROL=n` in your `prj.conf`.

QEMU You can run the Zephyr Host on the *QEMU emulator* and have it interact with a physical external Bluetooth Controller. Refer to *Running on QEMU and Native POSIX* for full instructions on how to build and run an application in this setup.

Native POSIX

Note: This is currently only available on GNU/Linux

The Native POSIX target builds your Zephyr application with the Zephyr kernel, and some minimal HW emulation as a native Linux executable. This executable is a normal Linux program, which can be debugged and instrumented like any other, and it communicates with a physical external Controller.

Refer to *Running on QEMU and Native POSIX* for full instructions on how to build and run an application in this setup.

Simulated nRF52 with BabbleSim

Note: This is currently only available on GNU/Linux

The nrf52_bsim board, is a simulated target board which emulates the necessary peripherals of a nrf52 SOC to be able to develop and test BLE applications. This board, uses:

- BabbleSim to simulate the nrf52 modem and the radio environment.
- The POSIX arch to emulate the processor.
- Models of the nrf52 HW

Just like with the `native_posix` target, the build result is a normal Linux executable. You can find more information on how to run simulations with one or several devices in this board's documentation

Currently, only *Combined builds* are possible, as this board does not yet have any models of a UART, or USB which could be used for an HCI interface towards another real or simulated device.

Initialization

The Bluetooth subsystem is initialized using the `bt_enable()` function. The caller should ensure that function succeeds by checking the return code for errors. If a function pointer is passed to `bt_enable()`, the initialization happens asynchronously, and the completion is notified through the given function.

Bluetooth Application Example

A simple Bluetooth beacon application is shown below. The application initializes the Bluetooth Subsystem and enables non-connectable advertising, effectively acting as a Bluetooth Low Energy broadcaster.

```

/*
 * Set Advertisement data. Based on the Eddystone specification:
 * https://github.com/google/eddystone/blob/master/protocol-specification.md
```

(continues on next page)

(continued from previous page)

```
 * https://github.com/google/eddystone/tree/master/eddystone-url
 */
static const struct bt_data ad[] = {
        BT_DATA_BYTES(BT_DATA_FLAGS, BT_LE_AD_NO_BREDR),
        BT_DATA_BYTES(BT_DATA_UUID16_ALL, 0xaa, 0xfe),
        BT_DATA_BYTES(BT_DATA_SVC_DATA16,
                      0xaa, 0xfe, /* Eddystone UUID */
                      0x10, /* Eddystone-URL frame type */
                      0x00, /* Calibrated Tx power at 0m */
                      0x00, /* URL Scheme Prefix http://www. */
                      'z', 'e', 'p', 'h', 'y', 'r',
                      'p', 'r', 'o', 'j', 'e', 'c', 't',
                      0x08) /* .org */
};

/* Set Scan Response data */
static const struct bt_data sd[] = {
        BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};

static void bt_ready(int err)
{
        char addr_s[BT_ADDR_LE_STR_LEN];
        bt_addr_le_t addr = {0};
        size_t count = 1;

        if (err) {
                printk("Bluetooth init failed (err %d)\n", err);
                return;
        }

        printk("Bluetooth initialized\n");

        /* Start advertising */
        err = bt_le_adv_start(BT_LE_ADV_NCONN_IDENTITY, ad, ARRAY_SIZE(ad),
                              sd, ARRAY_SIZE(sd));
        if (err) {
                printk("Advertising failed to start (err %d)\n", err);
                return;
        }


        /* For connectable advertising you would use
         * bt_le_oob_get_local().  For non-connectable non-identity
         * advertising an non-resolvable private address is used;
         * there is no API to retrieve that.
         */

        bt_id_get(&addr, &count);
        bt_addr_le_to_str(&addr, addr_s, sizeof(addr_s));

        printk("Beacon started, advertising as %s\n", addr_s);
}

void main(void)
{
```

(continues on next page)

(continued from previous page)

```
61          int err;
62
63          printk("Starting Beacon Demo\n");
64
65          /* Initialize the Bluetooth Subsystem */
66          err = bt_enable(bt_ready);
67          if (err) {
68                  printk("Bluetooth init failed (err %d)\n", err);
69          }
70  }
```

The key APIs employed by the beacon sample are *bt_enable()* that's used to initialize Bluetooth and then *bt_le_adv_start()* that's used to start advertising a specific combination of advertising and scan response data.

8.3.6 AutoPTS on Windows 10 with nRF52 board

Overview

This tutorial shows how to setup AutoPTS client and server to run both on Windows 10. We use WSL1 with Ubuntu only to build a Zephyr project to an elf file, because Zephyr SDK is not available on Windows yet. Tutorial covers only nrf52840dk.

Update Windows and drivers

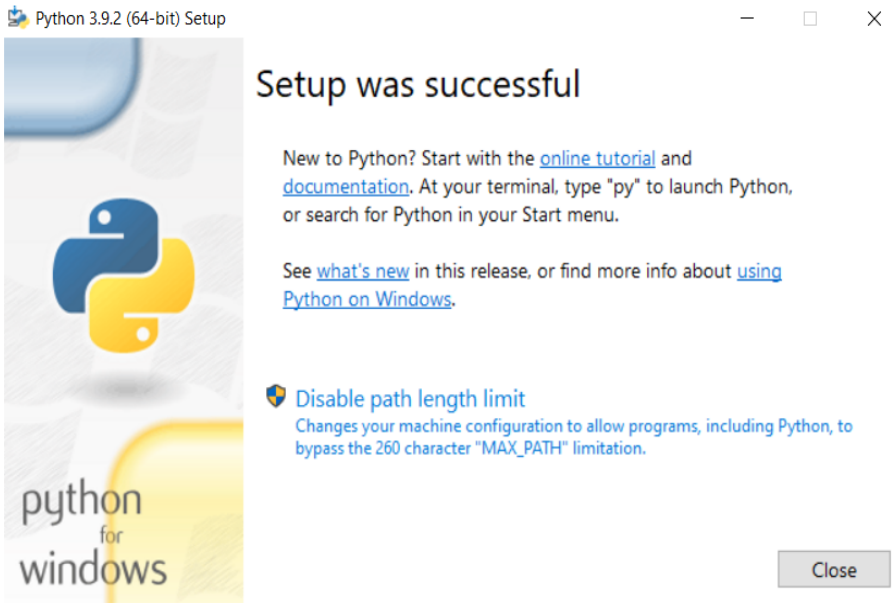
Update Windows in:

Start -> Settings -> Update & Security -> Windows Update

Update drivers, following the instructions from your hardware vendor.

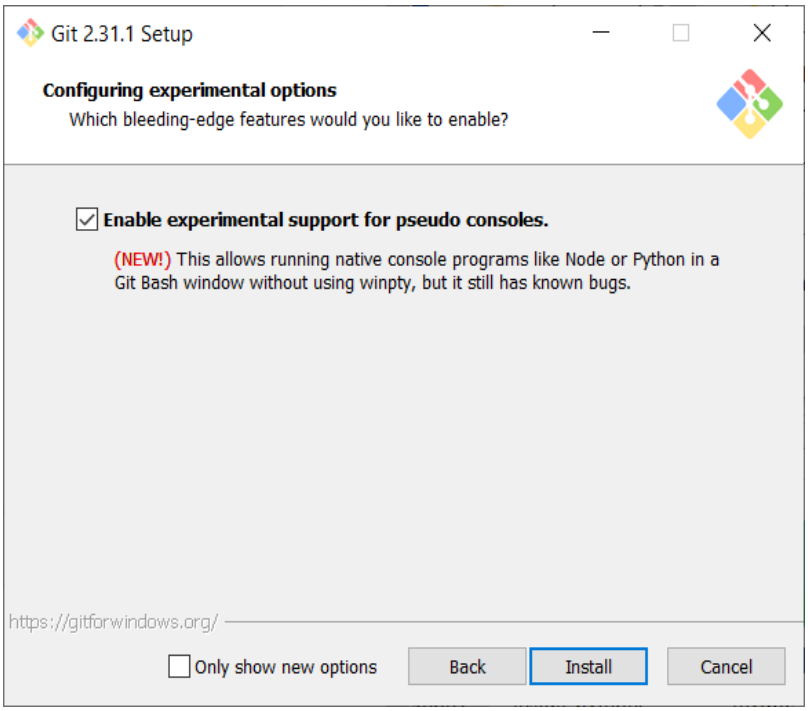
Install Python 3

Download and install Python 3. Setup was tested with versions >=3.8. Let the installer add the Python installation directory to the PATH and disable the path length limitation.



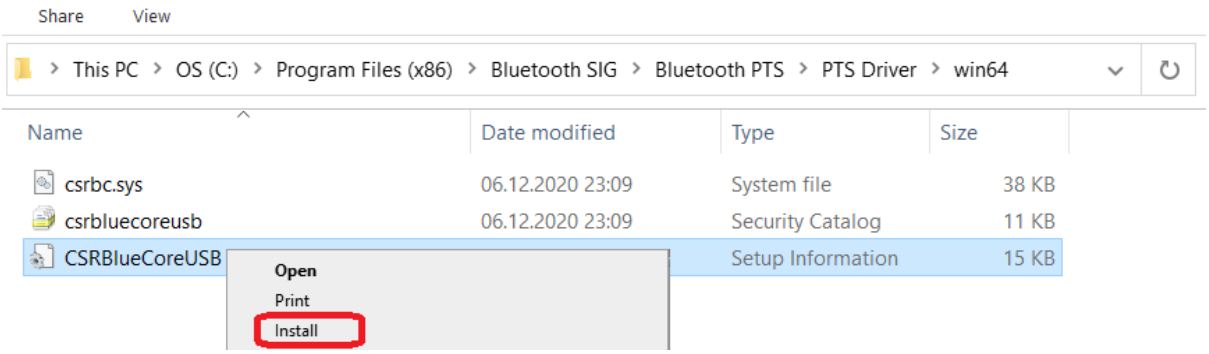
Install Git

Download and install Git. During installation enable option: Enable experimental support for pseudo consoles. We will use Git Bash as Windows terminal.



Install PTS 8

Install latest PTS from https://www.bluetooth.org. Remember to install drivers from installation directory "C:/Program Files (x86)/Bluetooth SIG/Bluetooth PTS/PTS Driver/win64/CSRBlueCoreUSB.inf"



Note: Starting with PTS 8.0.1 the Bluetooth Protocol Viewer is no longer included. So to capture Bluetooth events, you have to download it separately.

Setup Zephyr project for Windows

Setup from Zephyr site https://docs.zephyrproject.org/latest/getting_started/index.html:

Open Git Bash and go to home:

```
cd ~
```

Install west:

```
pip3 install west
```

Get the Zephyr source code:

```
west init zephyrproject
```

Go into freshly created folder:

```
cd zephyrproject
```

Run:

```
west update
```

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

```
west zephyr-export
```

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

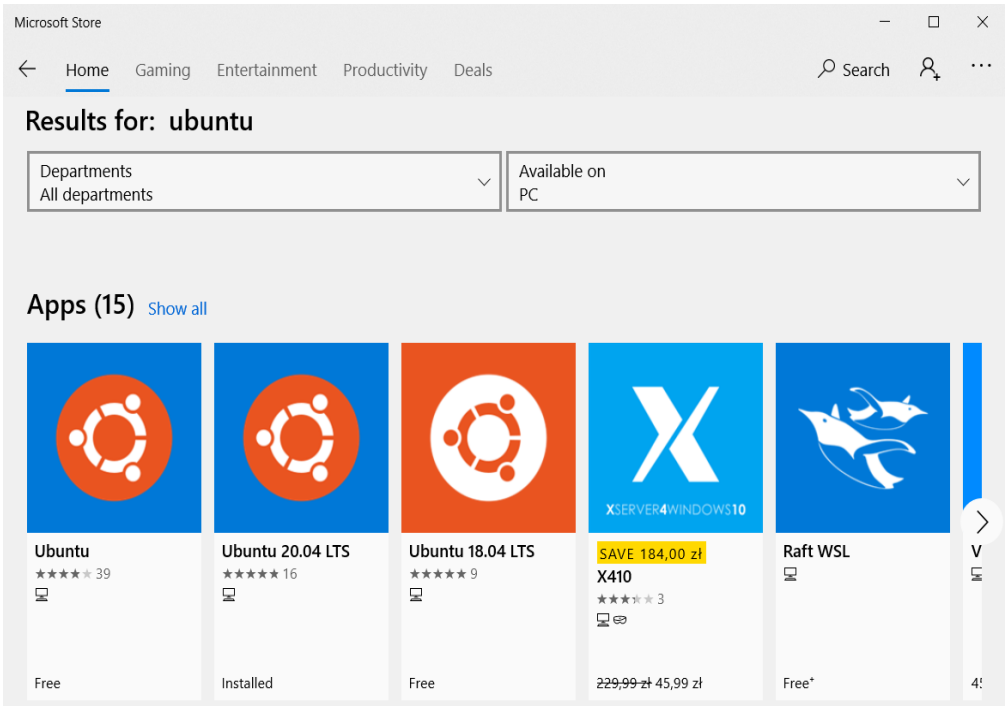
```
pip3 install -r ~\zephyrproject\zephyr\scripts\requirements.txt
```

Setup WSL1 with Ubuntu 20.4

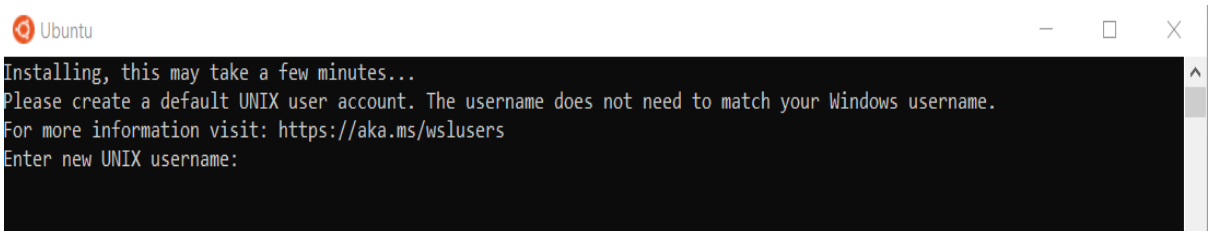
Setup Install Ubuntu 20.4 on WSL1. Open PowerShell as Administrator and run:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /
↪norestart
```

Restart Windows. After restart, open Microsoft Store and install Ubuntu 20.4 LTS.



Run Ubuntu. You will be asked to create a user account and password:



When finished, run commands:

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

Install python3:

```
sudo apt install python3
```

Install pip:

```
sudo apt install python3-pip
```

Install west:

```
pip3 install --user -U west
```

Export local bin to PATH:

```
echo 'export PATH=~/.local/bin:"$PATH"' >> ~/.bashrc
```

Reload PATH:

```
source ~/.bashrc
```

Install cmake:

```
sudo apt install cmake
```

Go to your zephyrproject:

```
cd /mnt/c/Users/Codecoup/zephyrproject
```

and then run:

```
west zephyr-export
pip3 install --user wheel
pip3 install --user -r /mnt/c/Users/codecoup/zephyrproject/zephyr/scripts/
↪requirements.txt
```

Check if all modules have been installed:

```
pip3 list
```

If modules still will be missing, just install them with:

```
pip3 install <module_name>
```

Install Ninja:

```
pip3 install ninja
```

Go to home:

```
cd ~
```

Download latest toolchain installer from https://github.com/zephyrproject-rtos/sdk-ng/releases. Move it to ~

```
mv /mnt/c/Users/Codecoup/Downloads/zephyr-sdk-<your_version>-setup.run ~
```

Give permissions to the installer:

```
chmod +x zephyr-sdk-<your_version>-setup.run
```

and run the installer:

```
./zephyr-sdk-<your_version>-setup.run -- -d ~/zephyr-sdk-<your_version>
```

Copy rules:

```
sudo cp ~/zephyr-sdk-<your_version>/sysroots/x86_64-pokysdk-linux/usr/share/openocd/
↪contrib/60-openocd.rules /etc/udev/rules.d
```

Restart the Ubuntu machine. You may want to shutdown all WSL consoles from Windows's Git Bash:

```
wsl --shutdown
```

After Ubuntu restart, go to:

```
cd /mnt/c/Users/codecoup/zephyrproject
```

and test if west can build:

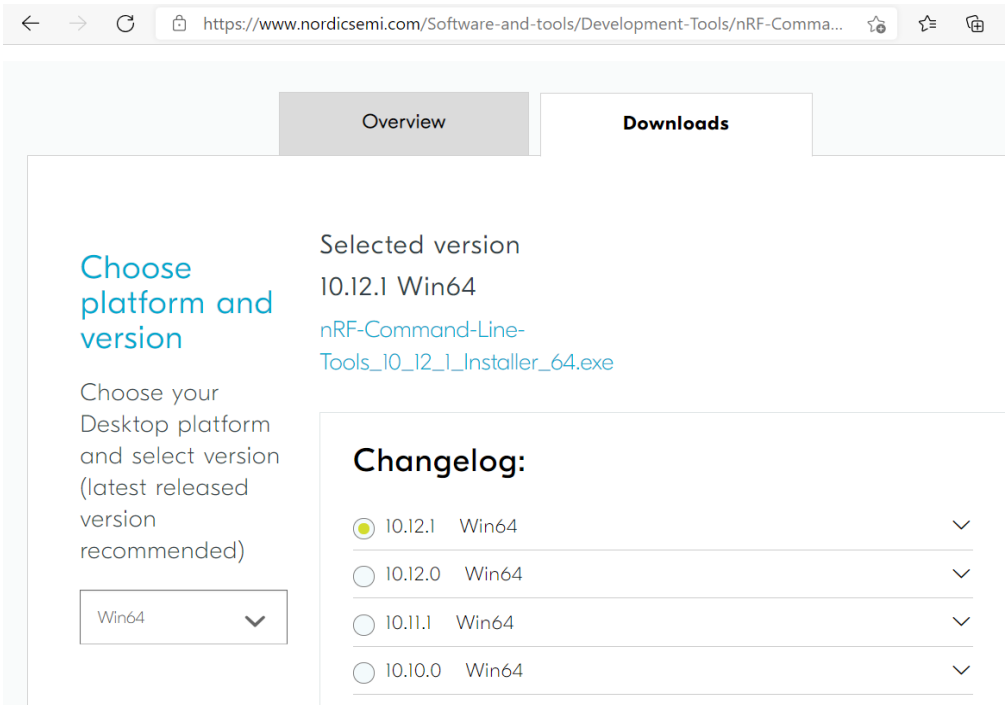
```
west build -p auto -b nrf52840dk_nrf52840 zephyr/tests/bluetooth/tester/
```

From now on, you can build projects by typing in Windows's Git Bash:

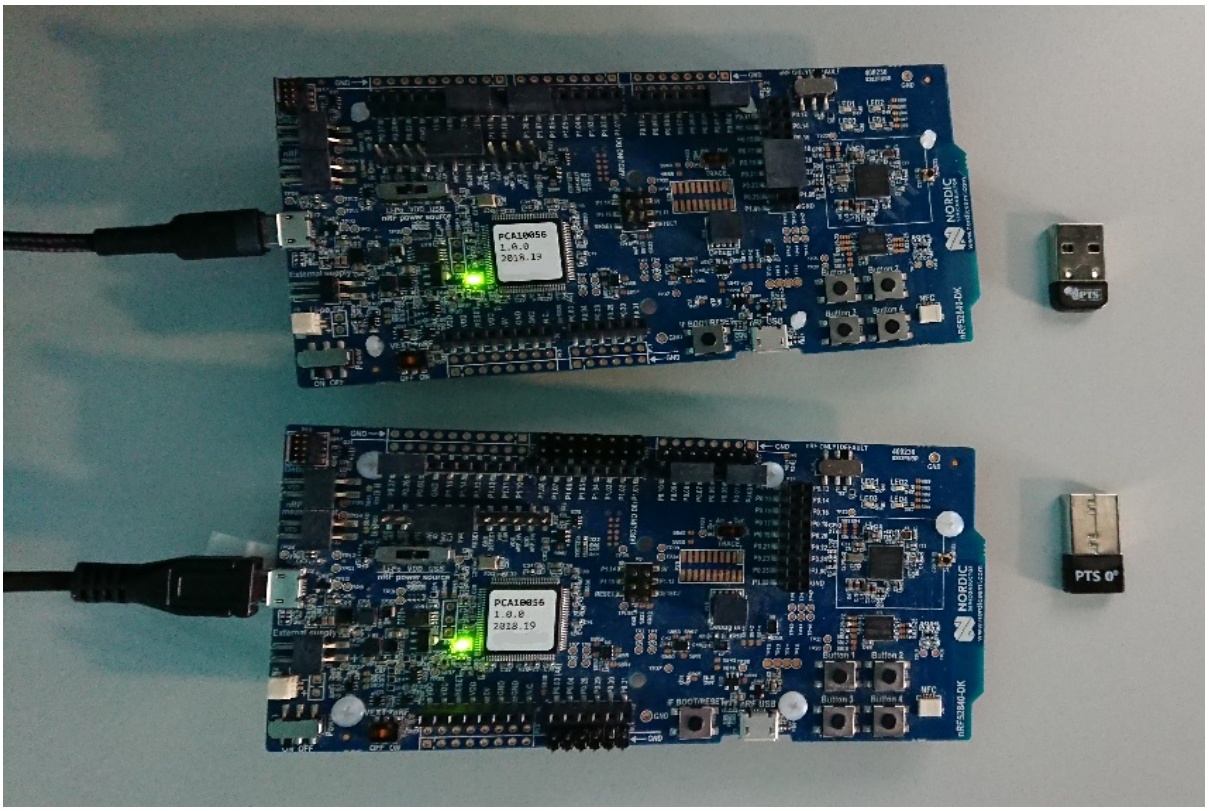
```
wsl -d Ubuntu-20.04 -u codecoup -- bash -c -i "cd /mnt/c/Users/Codecoup/zephyrproject/
↪ ; west build -p auto -b nrf52840dk_nrf52840 zephyr/tests/bluetooth/tester/"
```

Install nrftools

On Windows download latest nrftools (version >= 10.12.1) from site https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download and run default install.

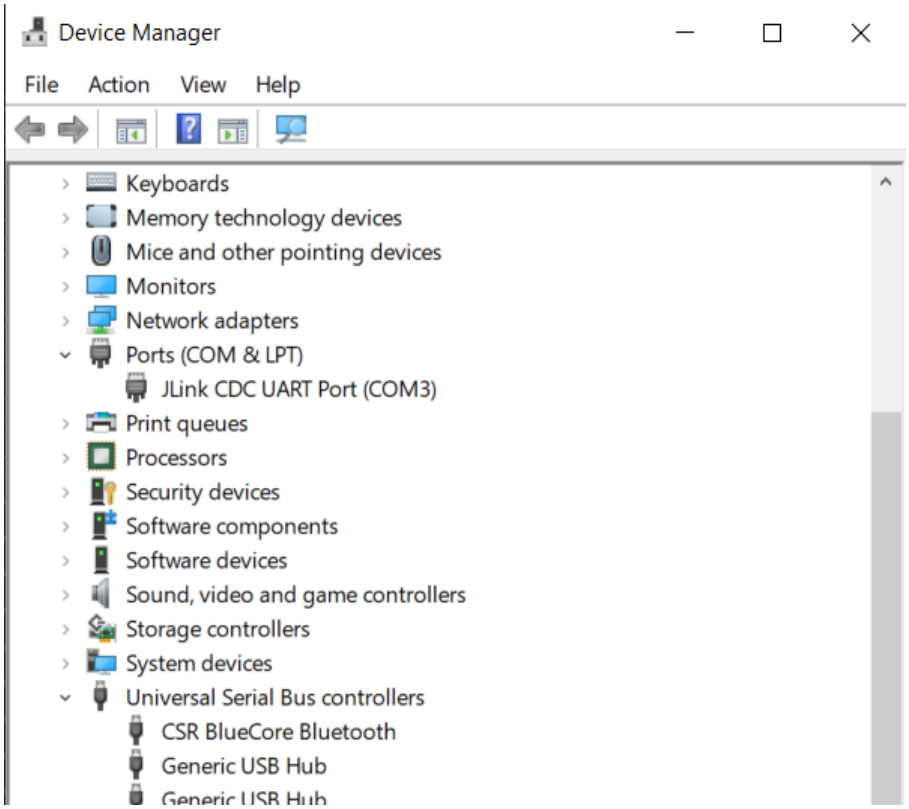


Connect devices



Flash board

In Device Manager find COM port of your nrf board. In my case it is COM3.



In Git Bash, go to zephyrproject

```
cd ~/zephyrproject
```

You can display flashing options with:

```
west flash --help
```

and flash board with built earlier elf file:

```
west flash --skip-rebuild --board-dir /dev/ttyS2 --elf-file ~/zephyrproject/build/
↪zephyr/zephyr.elf
```

Note that west does not accept COMs, so use /dev/ttyS2 as the COM3 equivalent, /dev/ttyS2 as the COM3 equivalent, etc.(/dev/ttyS + decremented COM number).

Setup auto-pts project

In Git Bash, clone project repo:

```
git clone https://github.com/intel/auto-pts.git
```

Go into the project folder:

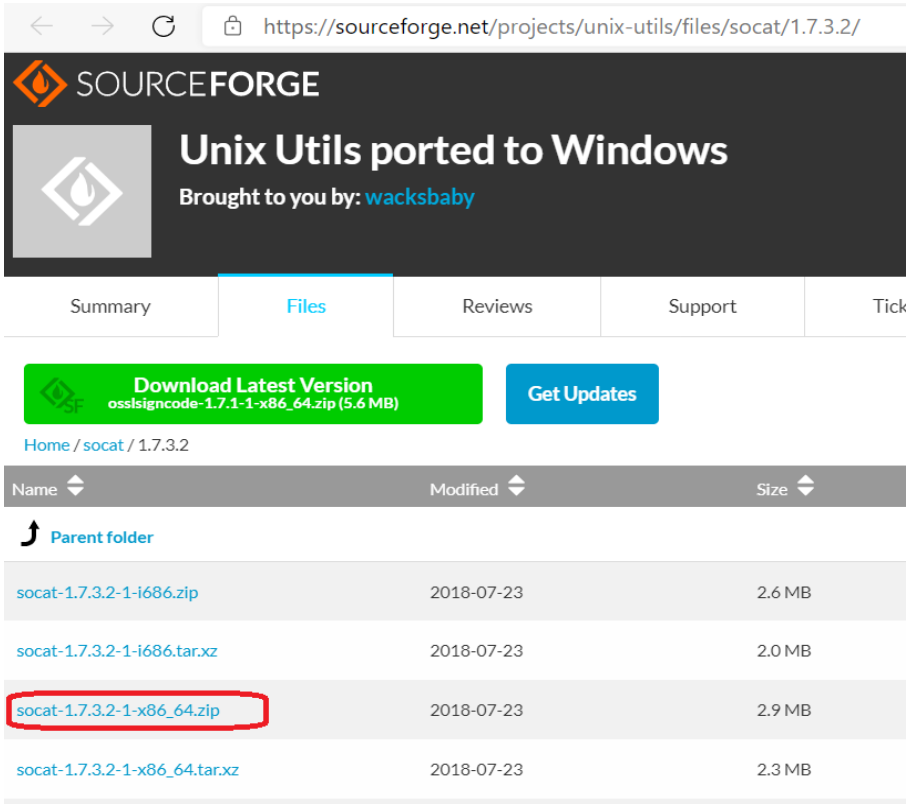
```
cd auto-pts
```

Install required python modules:

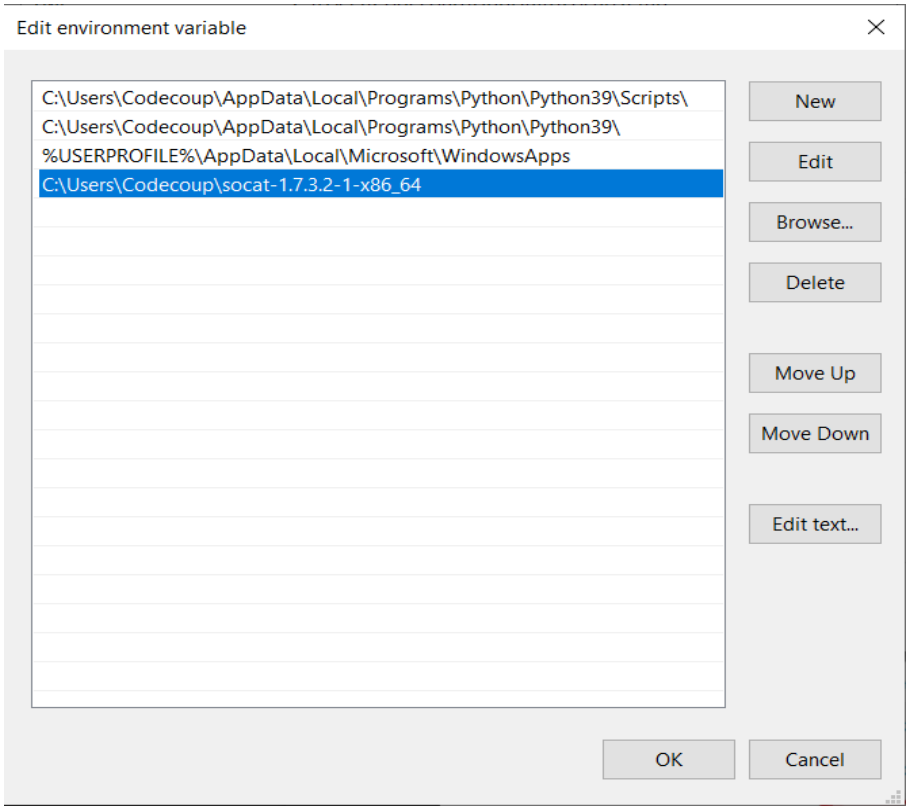
```
pip3 install --user wheel
pip3 install --user -r autoptsserver_requirements.txt
pip3 install --user -r autoptsclient_requirements.txt
```

Install socat.exe

Download and extract socat.exe from https://sourceforge.net/projects/unix-utils/files/socat/1.7.3.2/ into folder ~/socat-1.7.3.2-1-x86_64/.



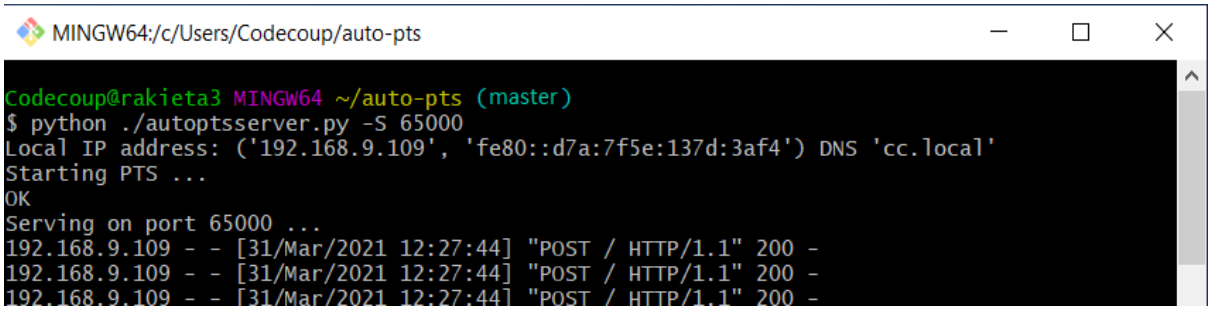
Add path to directory of socat.exe to PATH:



Running AutoPTS

Server and client by default will run on localhost address. Run server:

```
python ./autoptsserver.py -S 65000
```

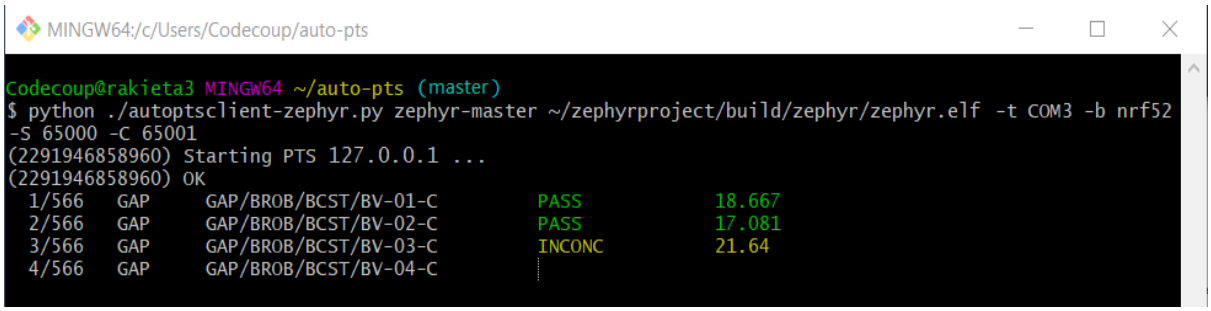


Note: If the error "ImportError: No module named pywintypes" appeared after the fresh setup, uninstall and install the pywin32 module:

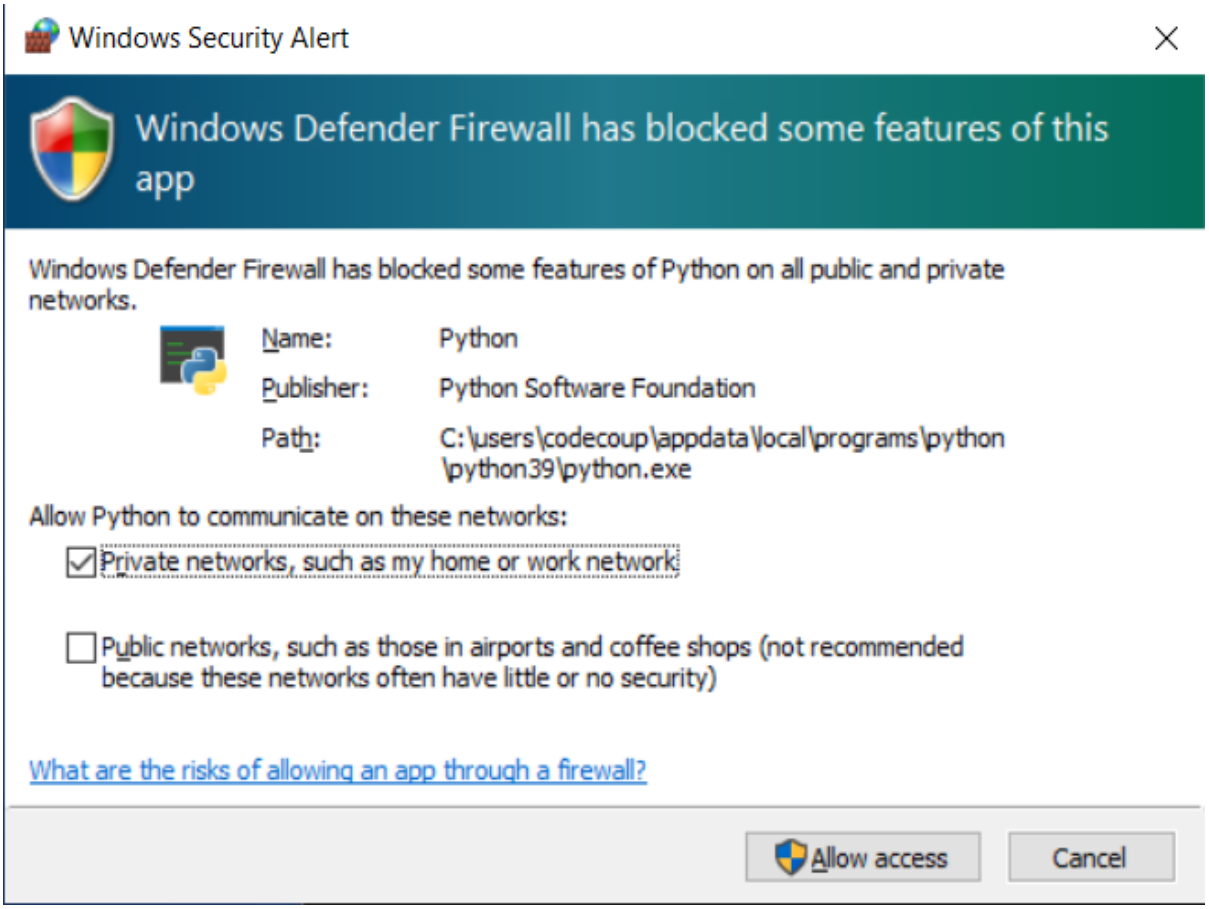
```
pip install --upgrade --force-reinstall pywin32
```

Run client:

```
python ./autoptsclient-zephyr.py zephyr-master ~/zephyrproject/build/zephyr/zephyr.
↪elf -t COM3 -b nrf52 -S 65000 -C 65001
```



At the first run, when Windows asks, enable connection through firewall:



Troubleshooting

- "When running actual hardware test mode, I have only BTP TIMEOUTs."

This is a problem with connection between auto-pts client and board. There are many possible causes. Try:

- Clean your auto-pts and zephyr repos with

Warning: This command will force the irreversible removal of all uncommitted files in the repo.

```
git clean -fdx
```

then build and flash tester elf again.

- If you have set up Windows on virtual machine, check if guest extensions are installed properly or change USB compatibility mode in VM settings to USB 2.0.
- Check, if firewall in not blocking python.exe or socat.exe.
- Check if board sends ready event after restart (hex 00 00 80 ff 00 00). Open serial connection to board with e.g. PuTTy with proper COM and baud rate. After board reset you should see some strings in console.
- Check if socat.exe creates tunel to board. Run in console

```
socat.exe -x -v tcp-listen:65123 /dev/ttyS2,raw,b115200
```

where /dev/ttyS2 is the COM3 equivalent. Open PuTTY, set connection type to Raw, IP to 127.0.0.1, port to 65123. After board reset you should see some strings in console.

8.3.7 AutoPTS on Linux

Overview

This tutorial shows how to setup AutoPTS client on Linux with AutoPTS server running on Windows 10 virtual machine. Tested with Ubuntu 20.4 and Linux Mint 20.4.

Supported methods to test zephyr bluetooth host:

- Testing Zephyr Host Stack on QEMU
- Testing Zephyr Host Stack on native posix
- Testing Zephyr combined (controller + host) build on Real hardware (such as nRF52)

For running with QEMU or native posix, please visit: https://docs.zephyrproject.org/latest/guides/bluetooth/bluetooth-tools.html?highlight=hci_uart#running-on-qemu-and-native-posix

Setup Linux

Setup Zephyr project Do the setup from Zephyr site https://docs.zephyrproject.org/latest/getting_started/index.html, especially:

Update OS This guide covers Ubuntu version 18.04 LTS and later.

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

Install dependencies

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

Get Zephyr and install Python dependencies Install west, and make sure ~/.local/bin is on your PATH environment variable:

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

Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

Install a Toolchain A toolchain provides a compiler, assembler, linker, and other programs required to build Zephyr applications.

Download the latest SDK installer from https://github.com/zephyrproject-rtos/sdk-ng/releases and run the installer, installing the SDK in ~/zephyr-sdk-<your_version>, e.g.:

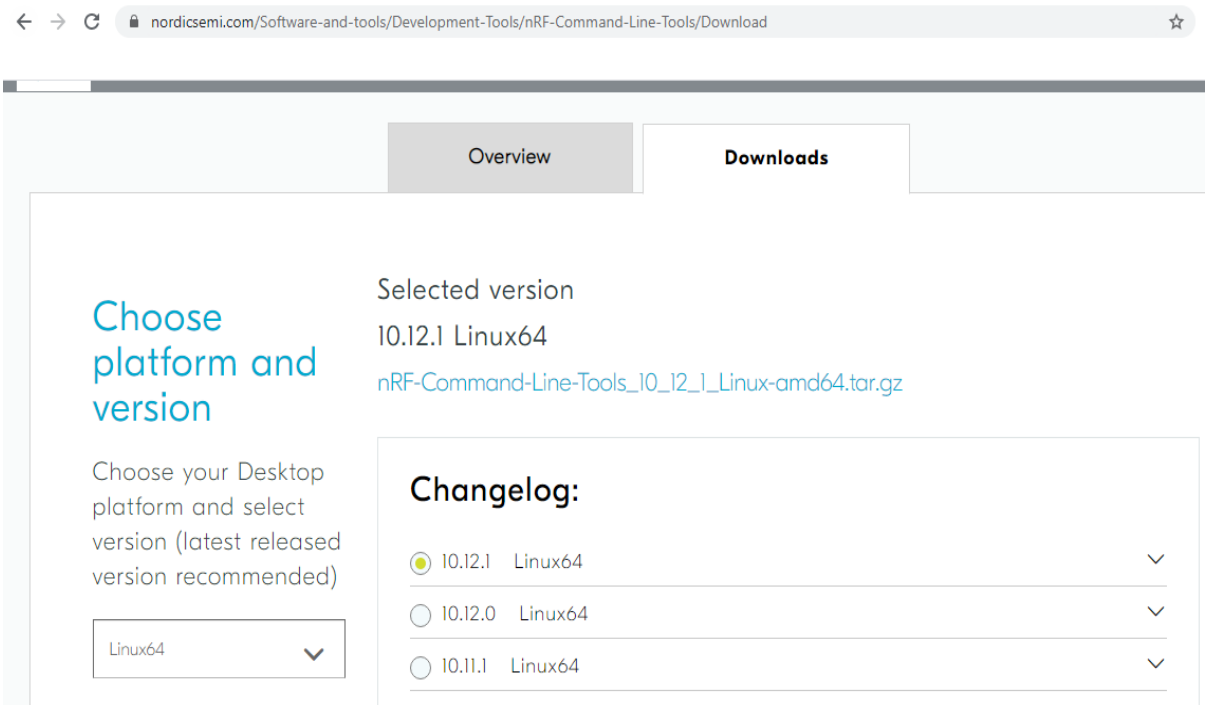
```
chmod +x zephyr-sdk-<your_version>-setup.run
./zephyr-sdk-<your_version>-setup.run -- -d ~/zephyr-sdk-<your_version>
```

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

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

Install nrftools (only required in the actual hardware test mode)

Download latest nrftools (version >= 10.12.1) from site https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download.



After you extract archive, you will see 2 .deb files, e.g.:

- JLink_Linux_V688a_x86_64.deb
- nRF-Command-Line-Tools_10_12_1_Linux-amd64.deb

and README.md. To install the tools, double click on each .deb file or fallow instructions from README.md.

Setup Windows 10 virtual machine

Choose and install your hypervisor like VMWare Workstation(preferred) or VirtualBox. On VirtualBox could be some issues, if your host has fewer than 6 CPU.

Create Windows virtual machine instance. Make sure it has at least 2 cores and installed guest extensions.

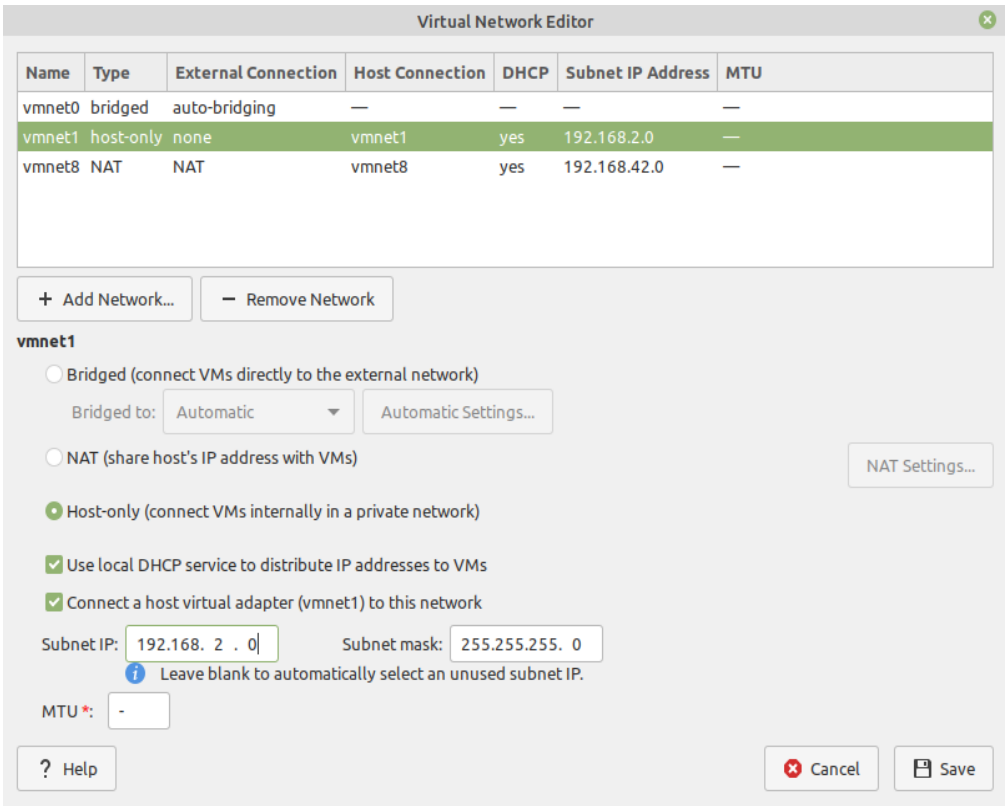
Setup tested with VirtualBox 6.1.18 and VMWare Workstation 16.1.1 Pro.

Update Windows Update Windows in:

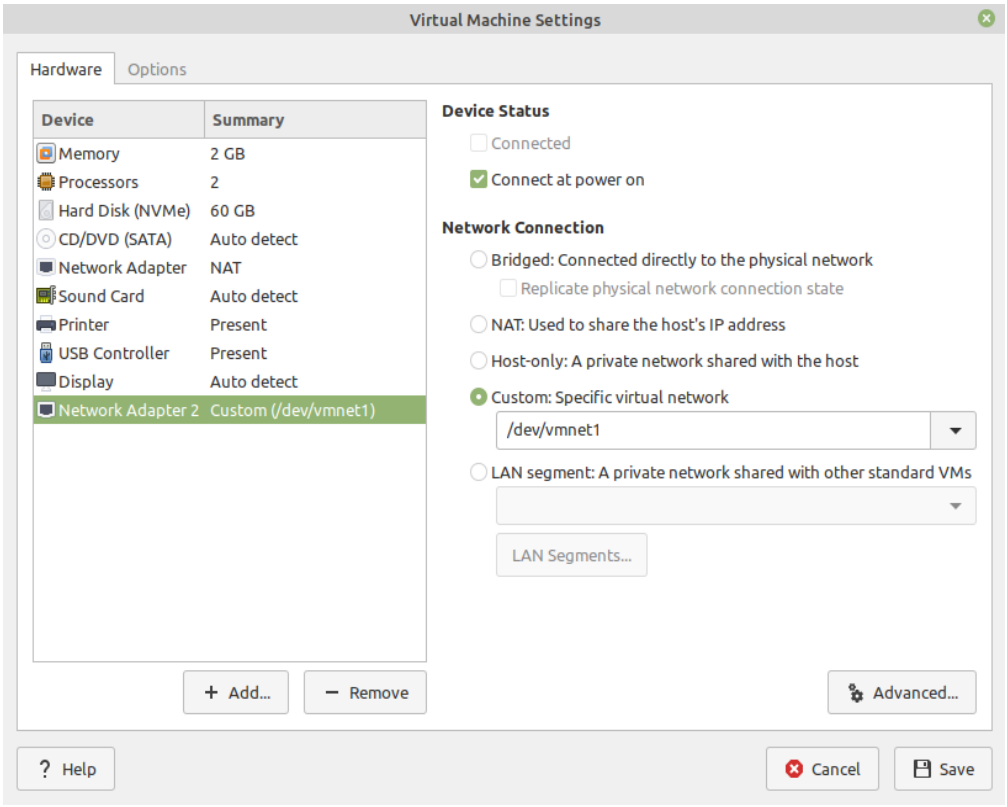
Start -> Settings -> Update & Security -> Windows Update

Setup static IP

WMWare Works On Linux, open Virtual Network Editor app and create network:



Open virtual machine network settings. Add custom adapter:



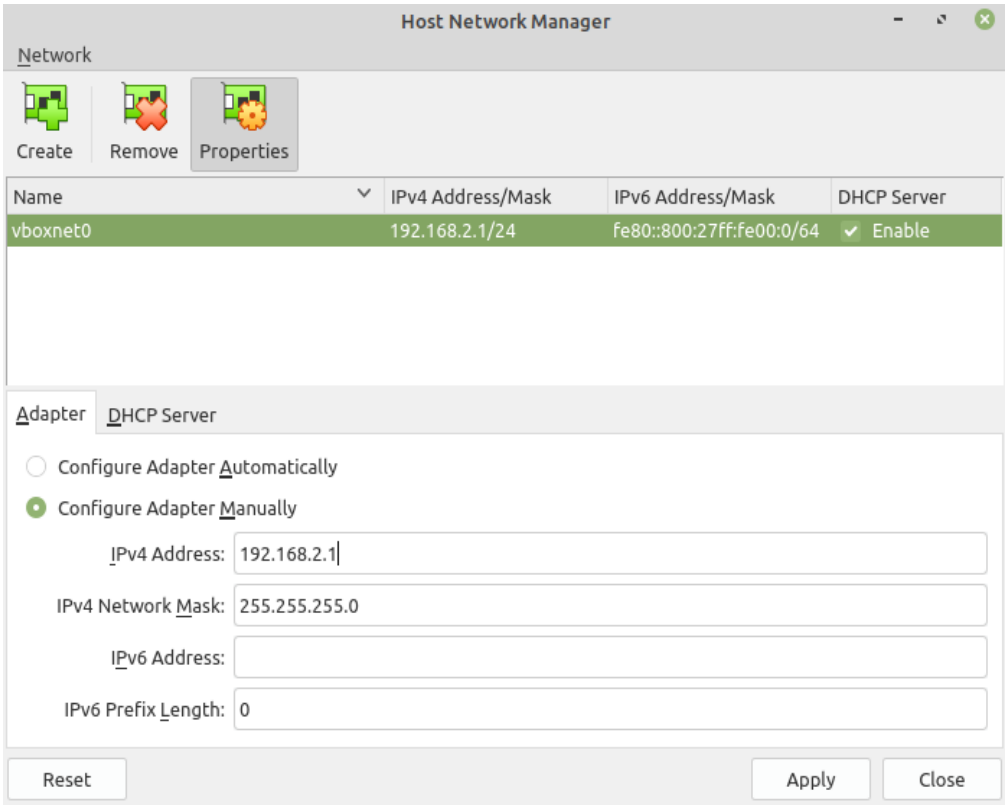
If you type 'ifconfig' in terminal, you should be able to find your host IP:

```
vmnet1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.1  netmask 255.255.255.0  broadcast 192.168.2.255
        inet6 fe80::250:56ff:fec0:1  prefixlen 64  scopeid 0x20<link>
        ether 00:50:56:c0:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 42  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

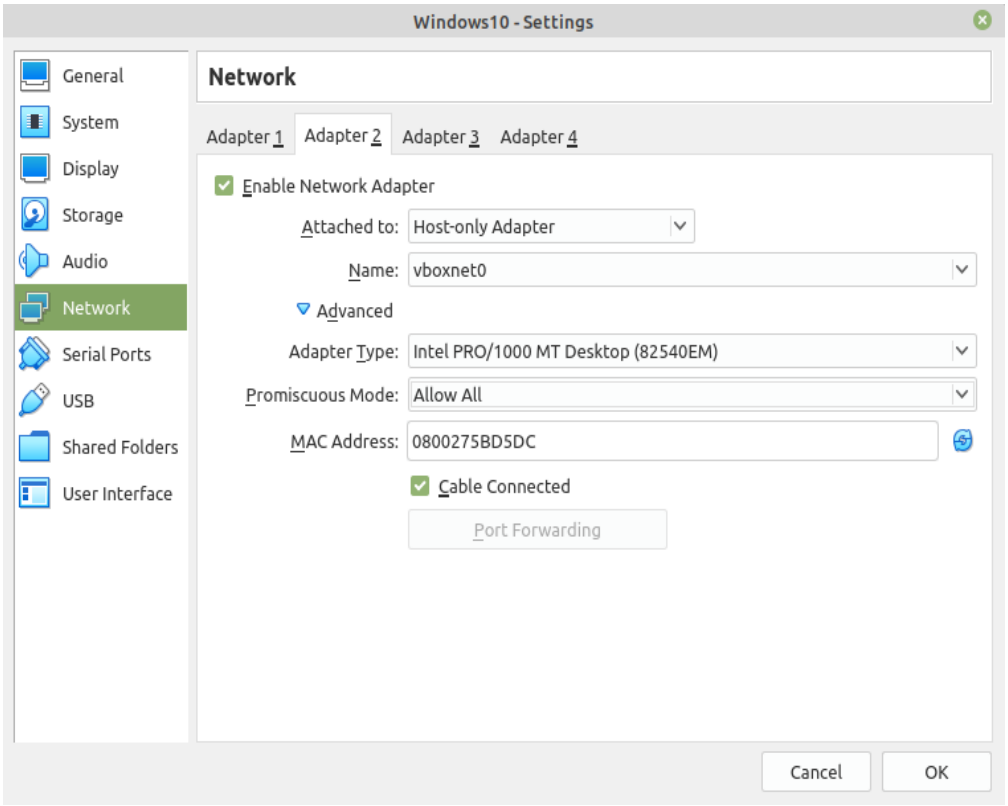
VirtualBox Go to:

File -> Host Network Manager

and create network:



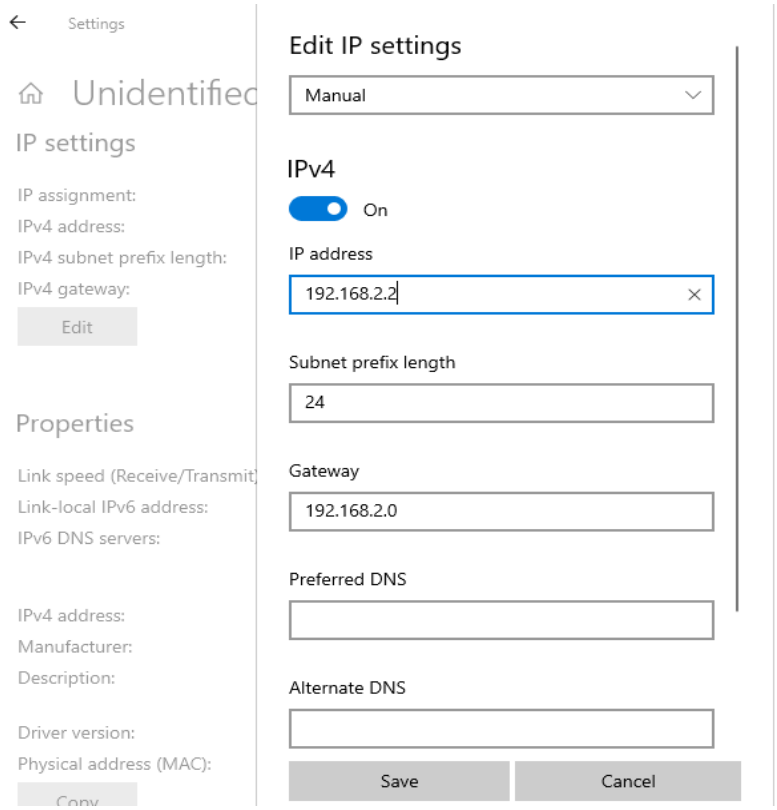
Open virtual machine network settings. On adapter 1 you will have created by default NAT. Add adapter 2:



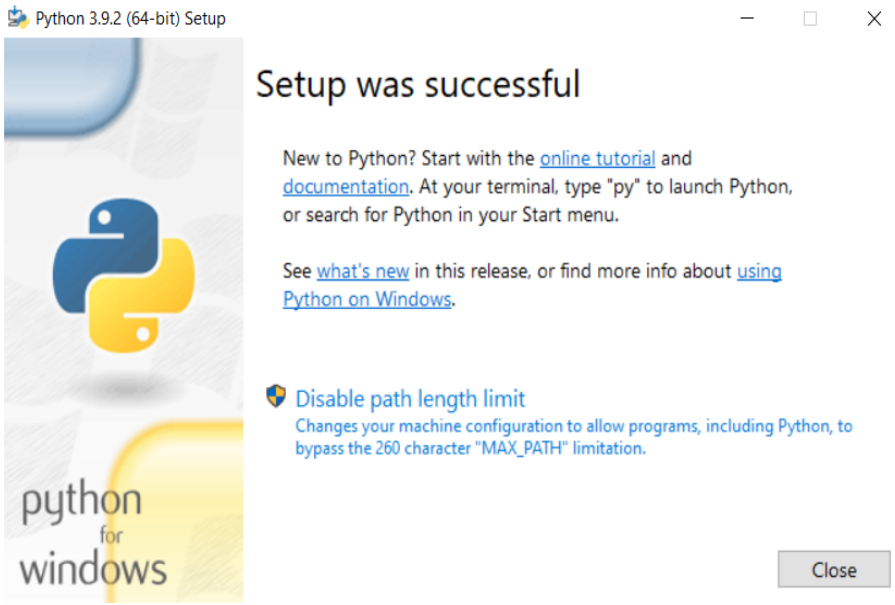
Windows Setup static IP on Windows virtual machine. Go to

Settings -> Network & Internet -> Ethernet -> Unidentified network -> Edit

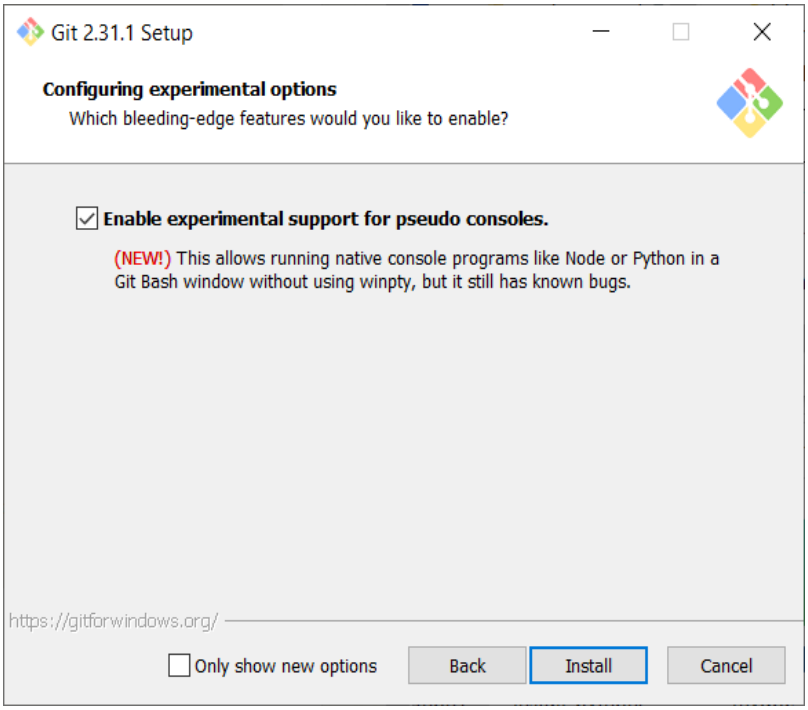
and set:



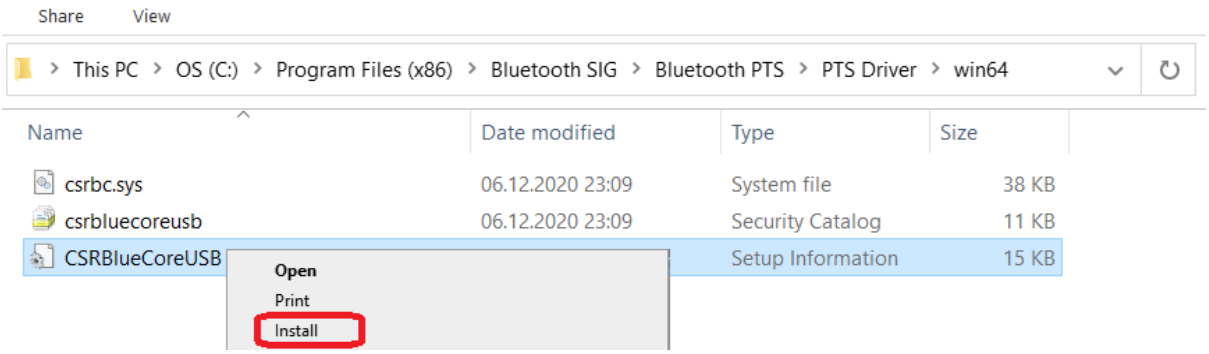
Install Python 3 Download and install latest Python 3 on Windows. Let the installer add the Python installation directory to the PATH and disable the path length limitation.



Install Git Download and install Git. During installation enable option: Enable experimental support for pseudo consoles. We will use Git Bash as Windows terminal.



Install PTS 8 On Windows virtual machine, install latest PTS from https://www.bluetooth.org. Remember to install drivers from installation directory “C:/Program Files (x86)/Bluetooth SIG/Bluetooth PTS/PTS Driver/win64/CSRBlueCoreUSB.inf”



Note: Starting with PTS 8.0.1 the Bluetooth Protocol Viewer is no longer included. So to capture Bluetooth events, you have to download it separately.

Connect PTS dongle With VirtualBox there should be no problem. Just find dongle in Devices -> USB and connect.

With VMWare you might need to use some trick, if you cannot find dongle in VM -> Removable Devices. Type in Linux terminal:

```
usb-devices
```

and find in output your PTS Bluetooth USB dongle

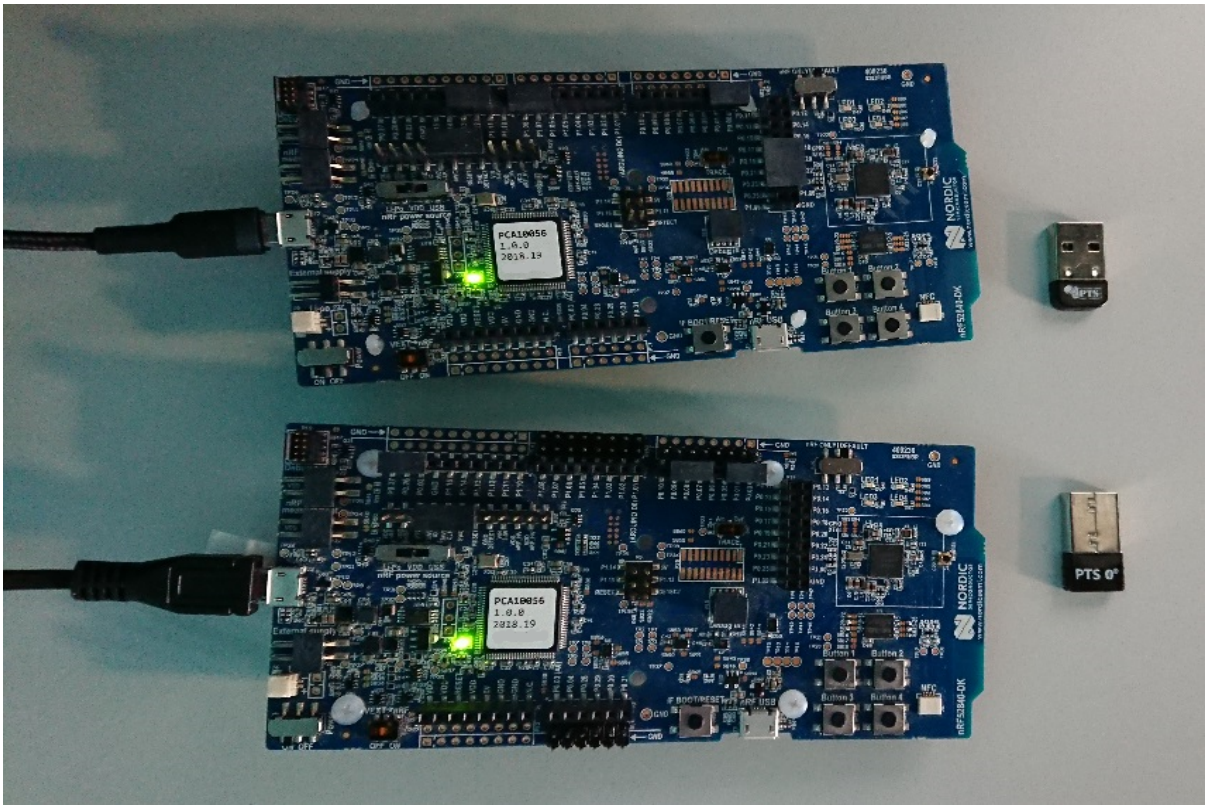
```
T:  Bus=01 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  6 Spd=12  MxCh= 0
D:  Ver= 2.00 Cls=e0(wlcon) Sub=01 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=0a12 ProdID=0001 Rev=25.20
S:  Product=CSR rck PTS Dongle
C:  #Ifs= 3 Cfg#= 1 Atr=c0 MxPwr=0mA
```

Note Vendor and ProdID number. Close VMWare Workstation and open .vmx of your virtual machine (path similar to /home/codecoup/vmware/Windows 10/Windows 10.vmx) in text editor. Write anywhere in the file following line:

```
usb.autoConnect.device0 = "0x0a12:0x0001"
```

just replace 0x0a12 with Vendor number and 0x0001 with ProdID number you found earlier.

Connect devices (only required in the actual hardware test mode)



Flash board (only required in the actual hardware test mode)

On Linux, go to ~/zephyrproject. There should be already ~/zephyrproject/build directory. Flash board:

```
west flash
```

Setup auto-pts project

AutoPTS client on Linux Clone auto-pts project:

```
git clone https://github.com/intel/auto-pts.git
```

Install socat, that is used to transfer BTP data stream from UART's tty file:

```
sudo apt-get install python-setuptools socat
```

Install required python modules:

```
cd auto-pts
pip3 install --user wheel
pip3 install --user -r autoptsclient_requirements.txt
```

Autopts server on Windows virtual machine In Git Bash, clone auto-pts project repo:

```
git clone https://github.com/intel/auto-pts.git
```

Install required python modules:

```
cd auto-pts
pip3 install --user wheel
pip3 install --user -r autoptsserver_requirements.txt
```

Restart virtual machine.

Running AutoPTS

Server and client by default will run on localhost address. Run server:

```
python ./autoptsserver.py
```

```
codecoup@DESKTOP-5SFTBGB MINGW64 ~/auto-pts (master)
$ python ./autoptsserver.py
Local IP address: ('10.0.2.15', 'fe80::ccb2:6ca1:6368:342a') DNS 'home'
Local IP address: ('192.168.2.2', 'fe80::fd12:15d5:2ddb:9147') DNS None
Starting PTS ...
OK
Serving on port 65000 ...
```

Testing Zephyr Host Stack on QEMU:

```
# A Bluetooth controller needs to be mounted.
# For running with HCI UART, please visit: https://docs.zephyrproject.org/latest/
↪samples/bluetooth/hci_uart/README.html#bluetooth-hci-uart

python ./autoptsclient-zephyr.py "C:\Users\USER_NAME\Documents\Profile Tuning Suite\
↪PTS_PROJECT\PTS_PROJECT.pqw6" \
    ~/zephyrproject/build/zephyr/zephyr.elf -i SERVER_IP -l LOCAL_IP
```

Testing Zephyr Host Stack on native posix:

```
# A Bluetooth controller needs to be mounted.
# For running with HCI UART, please visit: https://docs.zephyrproject.org/latest/
↪samples/bluetooth/hci_uart/README.html#bluetooth-hci-uart

west build -b native_posix zephyr/tests/bluetooth/tester/ -DOVERLAY_CONFIG=overlay-
↪native.conf
```

(continues on next page)

(continued from previous page)

```
sudo python ./autoptsclient-zephyr.py "C:\Users\USER_NAME\Documents\Profile Tuning␣
↪Suite\PTS_PROJECT\PTS_PROJECT.pqw6" \
    ~/zephyrproject/build/zephyr/zephyr.exe -i SERVER_IP -l LOCAL_IP --hci 0
```

Testing Zephyr combined (controller + host) build on nRF52:

Note: If the error "ImportError: No module named pywintypes" appeared after the fresh setup, uninstall and install the pywin32 module:

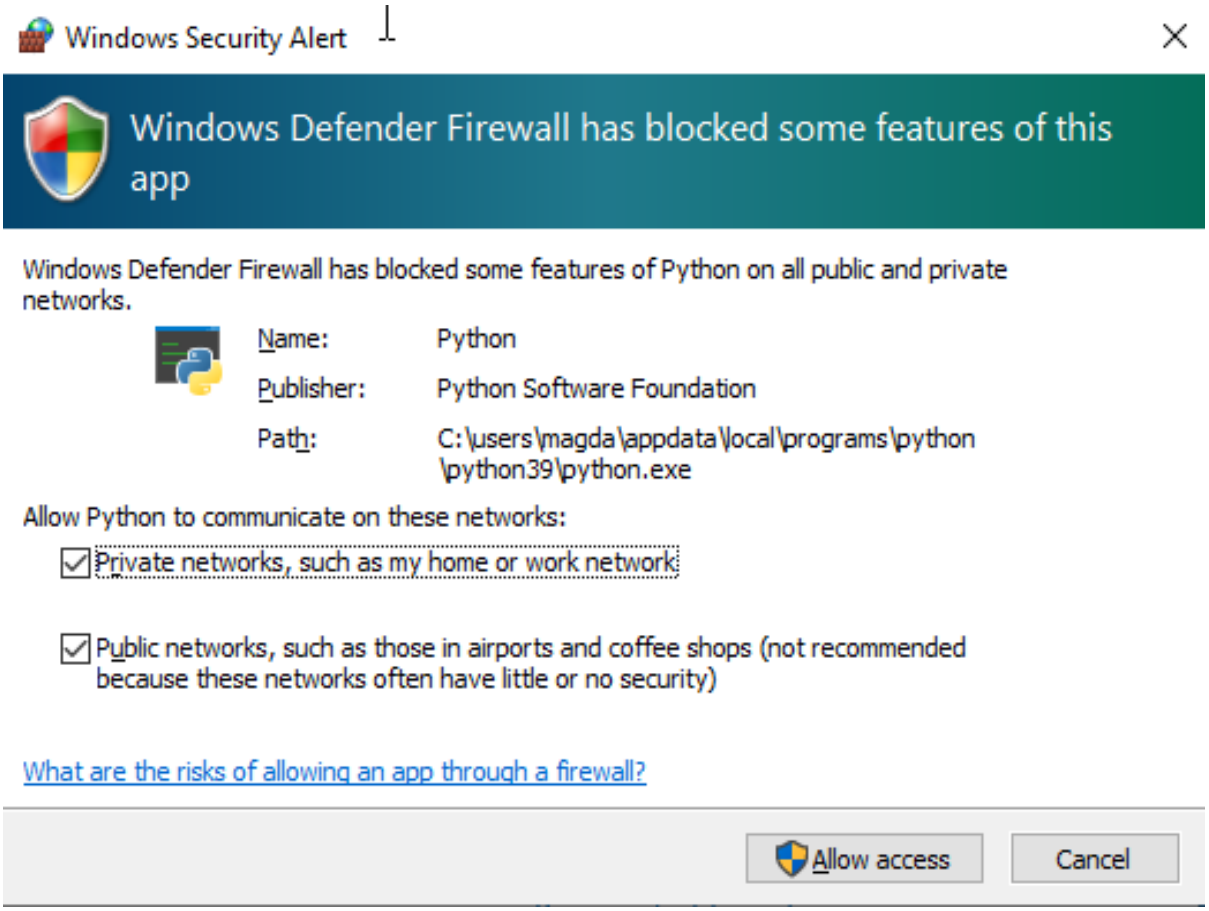
```
pip install --upgrade --force-reinstall pywin32
```

Run client:

```
python ./autoptsclient-zephyr.py zephyr-master ~/zephyrproject/build/zephyr/zephyr.
↪elf -t /dev/ACM0 \
    -b nrf52 -l 192.168.2.1 -i 192.168.2.2
```

```
codecoup@:~/auto-pts$ ./autoptsclient-zephyr.py zephyr-master ~/zephyrproject/build/zephyr/zephyr.elf
-t /dev/ttyACM0 -b nrf52 -l 192.168.2.1 -i 192.168.2.2
(140092026955280) Starting PTS 192.168.2.2 ...
(140092026955280) OK
  1/629   DIS     DIS/SR/SD/BV-01-C            PASS            23.477
  2/629   DIS     DIS/SR/DEC/BV-01-C           PASS            22.659
```

At the first run, when Windows asks, enable connection through firewall:



Troubleshooting

- "After running one test, I need to restart my Windows virtual machine to run another, because of fail verdict from APICOM in PTS logs."

It means your virtual machine has not enough processor cores or memory. Try to add more in settings. Note that a host with 4 CPUs could be not enough with VirtualBox as hypervisor. In this case, choose rather VMWare Workstation.

- "I cannot start autoptsserver-zephyr.py. I always got error:"

```
codecoup@DESKTOP-5SFTBGB MINGW64 ~/auto-pts (master)
$ python ./autoptsserver.py
Local IP address: ('10.0.2.15', 'fe80::ccb2:6ca1:6368:342a') DNS 'home'
Local IP address: ('192.168.2.2', 'fe80::fd12:15d5:2ddb:9147') DNS None
Starting PTS ...
Traceback (most recent call last):
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 127, in <module>
    main()
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 115, in main
    pts = PyPTSWithXmlRpcCallback()
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 52, in __init__
    ptscontrol.PyPTS.__init__(self)
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 267, in __init__
    self.restart_pts()
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 391, in restart_pts
    self.start_pts()
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 440, in start_pts
    log("PTS Bluetooth Address: %s", self.get_bluetooth_address())
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 762, in get_bluetooth_address
    raise e
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 759, in get_bluetooth_address
    address = self._pts.GetPTSBluetoothAddress()
  File "<COMObject ProfileTuningSuite_6.PTSControlServer>", line 3, in GetPTSBluetoothAddress
pywintypes.com_error: (-2147352567, 'Exception occurred.', (0, 'PTS', '(HRESULT:0x849C0043) Error HResult!', None
, 0, -2070151101), None)
```

One or more of the fallowing steps should help:

- Close all PTS Windows.
- Replug PTS bluetooth dongle.
- Delete temporary workspace. You will find it in auto-pts-code/workspaces/zephyr/zephyr-master/ as temp_zephyr-master. Be careful, do not remove the original one zephyr-master.pqw6.
- Restart Windows virtual machine.

8.4 Documentation Generation

These instructions will walk you through generating the Zephyr Project's documentation on your local system using the same documentation sources as we use to create the online documentation found at https://docs.zephyrproject.org

8.4.1 Documentation overview

Zephyr Project content is written using the reStructuredText markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted stand-alone website. Developers can view this content either in its raw form as .rst markup files, or you can generate the HTML content and view it with a web browser directly on your workstation. This same .rst content is also fed into the Zephyr Project's public website documentation area (with a different theme applied).

You can read details about reStructuredText, and Sphinx from their respective websites.

The project's documentation contains the following items:

- ReStructuredText source files used to generate documentation found at the https://docs.zephyrproject.org website. Most of the reStructuredText sources are found in the `/doc` directory,

but others are stored within the code source tree near their specific component (such as `/samples` and `/boards`)

- Doxygen-generated material used to create all API-specific documents also found at https://docs.zephyrproject.org
- Script-generated material for kernel configuration options based on Kconfig files found in the source code tree

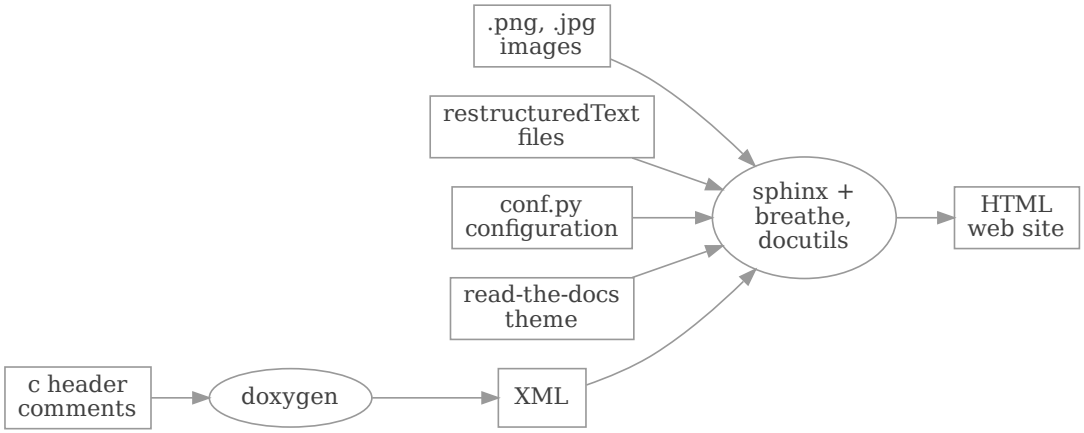


Fig. 4: Schematic of the documentation build process

The reStructuredText files are processed by the Sphinx documentation system, and make use of the breathe extension for including the doxygen-generated API material. Additional tools are required to generate the documentation locally, as described in the following sections.

8.4.2 Installing the documentation processors

Our documentation processing has been tested to run with:

- Doxygen version 1.8.13
- Graphviz 2.43
- Latexmk version 4.56
- All Python dependencies listed in the repository file `scripts/requirements-doc.txt`

In order to install the documentation tools, first install Zephyr as described in *Getting Started Guide*. Then install additional tools that are only required to generate the documentation, as described below:

Linux

On Ubuntu Linux:

```
sudo apt-get install --no-install-recommends doxygen graphviz librsvg2-bin \
texlive-latex-base texlive-latex-extra latexmk texlive-fonts-recommended
```

On Fedora Linux:

```
sudo dnf install doxygen graphviz texlive-latex latexmk \
texlive-collection-fontsrecommended librsvg2-tools
```

On Clear Linux:

```
sudo swupd bundle-add texlive graphviz
```

On Arch Linux:

```
sudo pacman -S graphviz doxygen librsvg texlive-core texlive-bin
```

macOS

Use `brew` and `tlmgr` to install the tools:

```
brew install doxygen graphviz mactex librsvg
tlmgr install latexmk
tlmgr install collection-fontsrecommended
```

Windows

Open a `cmd.exe` window as **Administrator** and run the following command:

```
choco install doxygen.install graphviz strawberryperl miktex rsvg-convert
```

Note: On Windows, the Sphinx executable `sphinx-build.exe` is placed in the `Scripts` folder of your Python installation path. Dependending on how you have installed Python, you might need to add this folder to your `PATH` environment variable. Follow the instructions in Windows Python Path to add those if needed.

8.4.3 Documentation presentation theme

Sphinx supports easy customization of the generated documentation appearance through the use of themes. Replace the theme files and do another `make htmldocs` and the output layout and style is changed. The `read-the-docs` theme is installed as part of the *Get Zephyr and install Python dependencies* step you took in the getting started guide.

8.4.4 Running the documentation processors

The `/doc` directory in your cloned copy of the Zephyr project git repo has all the .rst source files, extra tools, and Makefile for generating a local copy of the Zephyr project's technical documentation. Assuming the local Zephyr project copy is in a folder `zephyr` in your home folder, here are the commands to generate the html content locally:

```
# On Linux/macOS
cd ~/zephyr/doc
# On Windows
cd %userprofile%\zephyr\doc

# Use cmake to configure a Ninja-based build system:
cmake -GNinja -B_build .

# Enter the build directory
cd _build

# To generate HTML output, run ninja on the generated build system:
ninja html
# If you modify or add .rst files, run ninja again:
ninja html
```

(continues on next page)

(continued from previous page)

```
# To generate PDF output, run ninja on the generated build system:
ninja pdf
```

Warning: The documentation build system creates copies in the build directory of every .rst file used to generate the documentation, along with dependencies referenced by those .rst files.

This means that Sphinx warnings and errors refer to the **copies**, and **not the version-controlled original files in Zephyr**. Be careful to make sure you don't accidentally edit the copy of the file in an error message, as these changes will not be saved.

Depending on your development system, it will take up to 15 minutes to collect and generate the HTML content. When done, you can view the HTML output with your browser started at `doc/_build/html/index.html` and if generated, the PDF file is available at `doc/_build/pdf/zephyr.pdf`.

If you want to build the documentation from scratch just delete the contents of the build folder and run `cmake` and then `ninja` again.

Note: If you add or remove a file from the documentation, you need to re-run CMake.

On Unix platforms a convenience Makefile at the `doc` folder of the Zephyr repository can be used to build the documentation directly from there:

```
cd ~/zephyr/doc

# To generate HTML output
make html

# To generate PDF output
make pdf
```

8.4.5 Filtering expected warnings

There are some known issues with Sphinx/Breathe that generate Sphinx warnings even though the input is valid C code. While these issues are being considered for fixing we have created a Sphinx extension that allows to filter them out based on a set of regular expressions. The extension is named `zephyr.warnings_filter` and it is located at `doc/_extensions/zephyr/warnings_filter.py`. The warnings to be filtered out can be added to the `doc/known-warnings.txt` file.

The most common warning reported by Sphinx/Breathe is related to duplicate C declarations. This warning may be caused by different Sphinx/Breathe issues:

- Multiple declarations of the same object are not supported
- Different objects (e.g. a struct and a function) can not share the same name
- Nested elements (e.g. in a struct or union) can not share the same name

8.4.6 Developer-mode Document Building

Building the documentation for all the Kconfig options significantly adds to the total doc build time. When making and testing major changes to the documentation, we provide an option to temporarily stub-out the auto-generated configuration documentation so the doc build process runs much faster.

To enable this mode, set the following option when invoking cmake:

```
-DKCONFIG_TURBO_MODE=1
```

or invoke make with the following target:

```
cd ~/zephyr

# To generate HTML output without detailed Kconfig
make html-fast
```

8.4.7 Linking external Doxygen projects against Zephyr

External projects that build upon Zephyr functionality and wish to refer to Zephyr documentation in Doxygen (through the use of @ref), can utilize the tag file exported at zephyr.tag

Once downloaded, the tag file can be used in a custom `doxyfile.in` as follows:

```
TAGFILES = "/path/to/zephyr.tag=https://docs.zephyrproject.org/latest/doxygen/html/"
```

For additional information refer to Doxygen External Documentation.

8.5 Coccinelle

Coccinelle is a tool for pattern matching and text transformation that has many uses in kernel development, including the application of complex, tree-wide patches and detection of problematic programming patterns.

Note: Linux and macOS development environments are supported, but not Windows.

8.5.1 Getting Coccinelle

The semantic patches included in the kernel use features and options which are provided by Coccinelle version 1.0.0-rc11 and above. Using earlier versions will fail as the option names used by the Coccinelle files and `coccicheck` have been updated.

Coccinelle is available through the package manager of many distributions, e.g. :

- Debian
- Fedora
- Ubuntu
- OpenSUSE
- Arch Linux
- NetBSD
- FreeBSD

Some distribution packages are obsolete and it is recommended to use the latest version released from the Coccinelle homepage at http://coccinelle.lip6.fr/

Or from Github at:

https://github.com/coccinelle/coccinelle

Once you have it, run the following commands:

```
./autogen
./configure
make
```

as a regular user, and install it with:

```
sudo make install
```

More detailed installation instructions to build from source can be found at:

https://github.com/coccinelle/coccinelle/blob/master/install.txt

8.5.2 Supplemental documentation

For Semantic Patch Language(SmPL) grammar documentation refer to:

http://coccinelle.lip6.fr/documentation.php

8.5.3 Using Coccinelle on Zephyr

`coccicheck` checker is the front-end to the Coccinelle infrastructure and has various modes:

Four basic modes are defined: `patch`, `report`, `context`, and `org`. The mode to use is specified by setting `--mode=<mode>` or `-m=<mode>`.

- `patch` proposes a fix, when possible.
- `report` generates a list in the following format: file:line:column-column: message
- `context` highlights lines of interest and their context in a diff-like style.Lines of interest are indicated with `-`.
- `org` generates a report in the Org mode format of Emacs.

Note that not all semantic patches implement all modes. For easy use of Coccinelle, the default mode is `report`.

Two other modes provide some common combinations of these modes.

- `chain` tries the previous modes in the order above until one succeeds.
- `rep+ctxt` runs successively the report mode and the context mode. It should be used with the C option (described later) which checks the code on a file basis.

8.5.4 Examples

To make a report for every semantic patch, run the following command:

```
./scripts/coccicheck --mode=report
```

To produce patches, run:

```
./scripts/coccicheck --mode=patch
```

The `coccicheck` target applies every semantic patch available in the sub-directories of `scripts/coccinelle` to the entire source code tree.

For each semantic patch, a commit message is proposed. It gives a description of the problem being checked by the semantic patch, and includes a reference to Coccinelle.

As any static code analyzer, Coccinelle produces false positives. Thus, reports must be carefully checked, and patches reviewed.

To enable verbose messages set `--verbose=1` option, for example:

```
./scripts/coccicheck --mode=report --verbose=1
```

8.5.5 Coccinelle parallelization

By default, `coccicheck` tries to run as parallel as possible. To change the parallelism, set the `--jobs=<number>` option. For example, to run across 4 CPUs:

```
./scripts/coccicheck --mode=report --jobs=4
```

As of Coccinelle 1.0.2 Coccinelle uses Ocaml parmap for parallelization, if support for this is detected you will benefit from parmap parallelization.

When parmap is enabled `coccicheck` will enable dynamic load balancing by using `--chunksize 1` argument, this ensures we keep feeding threads with work one by one, so that we avoid the situation where most work gets done by only a few threads. With dynamic load balancing, if a thread finishes early we keep feeding it more work.

When parmap is enabled, if an error occurs in Coccinelle, this error value is propagated back, the return value of the `coccicheck` command captures this return value.

8.5.6 Using Coccinelle with a single semantic patch

The option `--cocci` can be used to check a single semantic patch. In that case, the variable must be initialized with the name of the semantic patch to apply.

For instance:

```
./scripts/coccicheck --mode=report --cocci=<example.cocci>
```

or:

```
./scripts/coccicheck --mode=report --cocci=./path/to/<example.cocci>
```

8.5.7 Controlling which files are processed by Coccinelle

By default the entire source tree is checked.

To apply Coccinelle to a specific directory, pass the path of specific directory as an argument.

For example, to check `drivers/usb/` one may write:

```
./scripts/coccicheck --mode=patch drivers/usb/
```

The `report` mode is the default. You can select another one with the `--mode=<mode>` option explained above.

8.5.8 Debugging Coccinelle SmPL patches

Using `coccicheck` is best as it provides in the spatch command line include options matching the options used when we compile the kernel. You can learn what these options are by using verbose option, you could then manually run Coccinelle with debug options added.

Alternatively you can debug running Coccinelle against SmPL patches by asking for stderr to be redirected to stderr, by default stderr is redirected to /dev/null, if you'd like to capture stderr you can specify the `--debug=file.err` option to `coccicheck`. For instance:

```
rm -f cocci.err
./scripts/coccicheck --mode=patch --debug=cocci.err
cat cocci.err
```

Debugging support is only supported when using Coccinelle >= 1.0.2.

8.5.9 Additional Flags

Additional flags can be passed to spatch through the SPFLAGS variable. This works as Coccinelle respects the last flags given to it when options are in conflict.

```
./scripts/coccicheck --sp-flag="--use-glimpse"
```

Coccinelle supports idutils as well but requires coccinelle >= 1.0.6. When no ID file is specified coccinelle assumes your ID database file is in the file .id-utils.index on the top level of the kernel, coccinelle carries a script scripts/idutils_index.sh which creates the database with:

```
mkid -i C --output .id-utils.index
```

If you have another database filename you can also just symlink with this name.

```
./scripts/coccicheck --sp-flag="--use-idutils"
```

Alternatively you can specify the database filename explicitly, for instance:

```
./scripts/coccicheck --sp-flag="--use-idutils /full-path/to/ID"
```

Sometimes coccinelle doesn't recognize or parse complex macro variables due to insufficient definition. Therefore, to make it parsable we explicitly provide the prototype of the complex macro using the `---macro-file-builtins <headerfile.h>` flag.

The `<headerfile.h>` should contain the complete prototype of the complex macro from which spatch engine can extract the type information required during transformation.

For example:

`Z_SYSCALL_HANDLER` is not recognized by coccinelle. Therefore, we put its prototype in a header file, say for example `mymacros.h`.

```
$ cat mymacros.h
#define Z_SYSCALL_HANDLER int xxx
```

Now we pass the header file `mymacros.h` during transformation:

```
./scripts/coccicheck --sp-flag="---macro-file-builtins mymacros.h"
```

See `spatch --help` to learn more about spatch options.

Note that the `--use-glimpse` and `--use-idutils` options require external tools for indexing the code. None of them is thus active by default. However, by indexing the code with one of these tools, and according to the cocci file used, spatch could proceed the entire code base more quickly.

8.5.10 SmPL patch specific options

SmPL patches can have their own requirements for options passed to Coccinelle. SmPL patch specific options can be provided by providing them at the top of the SmPL patch, for instance:

```
// Options: --no-includes --include-headers
```

8.5.11 Proposing new semantic patches

New semantic patches can be proposed and submitted by kernel developers. For sake of clarity, they should be organized in the sub-directories of `scripts/coccinelle/`.

The cocci script should have the following properties:

- The script **must** have `report` mode.
- The first few lines should state the purpose of the script using `///` comments . Usually, this message would be used as the commit log when proposing a patch based on the script.

Example

```
/// Use ARRAY_SIZE instead of dividing sizeof array with sizeof an element
```

- A more detailed information about the script with exceptional cases or false positives (if any) can be listed using `//#` comments.

Example

```
//# This makes an effort to find cases where ARRAY_SIZE can be used such as
//# where there is a division of sizeof the array by the sizeof its first
//# element or by any indexed element or the element type. It replaces the
//# division of the two sizeofs by ARRAY_SIZE.
```

- Confidence: It is a property defined to specify the accuracy level of the script. It can be either `High`, `Moderate` or `Low` depending upon the number of false positives observed.

Example

```
// Confidence: High
```

- Virtual rules: These are required to support the various modes framed in the script. The virtual rule specified in the script should have the corresponding mode handling rule.

Example

```
virtual context

@depends on context@
type T;
T[] E;
@@
(
* (sizeof(E)/sizeof(*E))
|
* (sizeof(E)/sizeof(E[...]))
|
* (sizeof(E)/sizeof(T))
)
```

8.5.12 Detailed description of the `report` mode

`report` generates a list in the following format:

```
file:line:column-column: message
```

Example

Running:

```
./scripts/coccicheck --mode=report --cocci=scripts/coccinelle/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@r depends on (org || report)@
type T;
T[] E;
position p;
@@
(
(sizeof(E)@p /sizeof(*E))
|
(sizeof(E)@p /sizeof(E[...]))
|
(sizeof(E)@p /sizeof(T))
)

@script:python depends on report@
p << r.p;
@@

msg="WARNING: Use ARRAY_SIZE"
coccilib.report.print_report(p[0], msg)

</smpl>
```

This SmPL excerpt generates entries on the standard output, as illustrated below:

```
ext/hal/nxp/mcux/drivers/lpc/fsl_wwdt.c:66:49-50: WARNING: Use ARRAY_SIZE
ext/hal/nxp/mcux/drivers/lpc/fsl_ctimer.c:74:53-54: WARNING: Use ARRAY_SIZE
ext/hal/nxp/mcux/drivers/imx/fsl_dcp.c:944:45-46: WARNING: Use ARRAY_SIZE
```

8.5.13 Detailed description of the `patch` mode

When the `patch` mode is available, it proposes a fix for each problem identified.

Example

Running:

```
./scripts/coccicheck --mode=patch --cocci=scripts/coccinelle/misc/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@depends on patch@
type T;
T[] E;
@@
(
- (sizeof(E)/sizeof(*E))
+ ARRAY_SIZE(E)
|
- (sizeof(E)/sizeof(E[...]))
+ ARRAY_SIZE(E)
|
- (sizeof(E)/sizeof(T))
+ ARRAY_SIZE(E)
)

</smpl>
```

This SmPL excerpt generates patch hunks on the standard output, as illustrated below:

```
diff -u -p a/ext/lib/encoding/tinycbor/src/cborvalidation.c b/ext/lib/encoding/
↪tinycbor/src/cborvalidation.c
--- a/ext/lib/encoding/tinycbor/src/cborvalidation.c
+++ b/ext/lib/encoding/tinycbor/src/cborvalidation.c
@@ -325,7 +325,7 @@ static inline CborError validate_number(
static inline CborError validate_tag(CborValue *it, CborTag tag, int flags, int␣
↪recursionLeft)
{
  CborType type = cbor_value_get_type(it);
-    const size_t knownTagCount = sizeof(knownTagData) / sizeof(knownTagData[0]);
+    const size_t knownTagCount = ARRAY_SIZE(knownTagData);
   const struct KnownTagData *tagData = knownTagData;
   const struct KnownTagData * const knownTagDataEnd = knownTagData + knownTagCount;
```

8.5.14 Detailed description of the `context` mode

`context` highlights lines of interest and their context in a diff-like style.

Note: The diff-like output generated is NOT an applicable patch. The intent of the `context` mode is to highlight the important lines (annotated with minus, -) and gives some surrounding context lines around. This output can be used with the diff mode of Emacs to review the code.

Example

Running:

```
./scripts/coccicheck --mode=context --cocci=scripts/coccinelle/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@depends on context@
```

(continues on next page)

(continued from previous page)

```
type T;
T[] E;
@@
(
* (sizeof(E)/sizeof(*E))
|
* (sizeof(E)/sizeof(E[...]))
|
* (sizeof(E)/sizeof(T))
)

</smpl>
```

This SmPL excerpt generates diff hunks on the standard output, as illustrated below:

```
diff -u -p ext/lib/encoding/tinycbor/src/cborvalidation.c /tmp/nothing/ext/lib/
↪encoding/tinycbor/src/cborvalidation.c
--- ext/lib/encoding/tinycbor/src/cborvalidation.c
+++ /tmp/nothing/ext/lib/encoding/tinycbor/src/cborvalidation.c
@@ -325,7 +325,6 @@ static inline CborError validate_number(
static inline CborError validate_tag(CborValue *it, CborTag tag, int flags, int␣
↪recursionLeft)
{
  CborType type = cbor_value_get_type(it);
-    const size_t knownTagCount = sizeof(knownTagData) / sizeof(knownTagData[0]);
   const struct KnownTagData *tagData = knownTagData;
   const struct KnownTagData * const knownTagDataEnd = knownTagData + knownTagCount;
```

8.5.15 Detailed description of the `org` mode

`org` generates a report in the Org mode format of Emacs.

Example

Running:

```
./scripts/coccicheck --mode=org --cocci=scripts/coccinelle/misc/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@r depends on (org || report)@
type T;
T[] E;
position p;
@@
(
(sizeof(E)@p /sizeof(*E))
|
(sizeof(E)@p /sizeof(E[...]))
|
(sizeof(E)@p /sizeof(T))
)
```

(continues on next page)

(continued from previous page)

```
@script:python depends on org@
p << r.p;
@@
coccilib.org.print_todo(p[0], "WARNING should use ARRAY_SIZE")

</smpl>
```

This SmPL excerpt generates Org entries on the standard output, as illustrated below:

```
* TODO [[view:ext/lib/encoding/tinycbor/src/cborvalidation.c::face=ovl-
↪face1::linb=328::colb=52::cole=53][WARNING should use ARRAY_SIZE]]
```

8.5.16 Coccinelle Mailing List

Subscribe to the coccinelle mailing list:

- https://systeme.lip6.fr/mailman/listinfo/cocci

Archives:

- https://lore.kernel.org/cocci/
- https://systeme.lip6.fr/pipermail/cocci/

8.6 Code And Data Relocation

8.6.1 Overview

This feature will allow relocating .text, .rodata, .data, and .bss sections from required files and place them in the required memory region. The memory region and file are given to the *scripts/gen_relocate_app.py* script in the form of a string. This script is always invoked from inside cmake.

This script provides a robust way to re-order the memory contents without actually having to modify the code. In simple terms this script will do the job of `__attribute__((section("name")))` for a bunch of files together.

8.6.2 Details

The memory region and file are given to the *scripts/gen_relocate_app.py* script in the form of a string.

An example of such a string is: `SRAM2:/home/xyz/zephyr/samples/hello_world/src/main.c, SRAM1:/home/xyz/zephyr/samples/hello_world/src/main2.c`

This script is invoked with the following parameters: `python3 gen_relocate_app.py -i input_string -o generated_linker -c generated_code`

Kconfig `CONFIG_CODE_DATA_RELOCATION` option, when enabled in `prj.conf`, will invoke the script and do the required relocation.

This script also trigger the generation of `linker_relocate.ld` and `code_relocation.c` files. The `linker_relocate.ld` file creates appropriate sections and links the required functions or variables from all the selected files.

Note: The text section is split into 2 parts in the main linker script. The first section will have some info regarding vector tables and other debug related info. The second section will have the complete text

section. This is needed to force the required functions and data variables to the correct locations. This is due to the behavior of the linker. The linker will only link once and hence this text section had to be split to make room for the generated linker script.

The `code_relocation.c` file has code that is needed for initializing data sections, and a copy of the text sections (if XIP). Also this contains code needed for bss zeroing and for data copy operations from ROM to required memory type.

The procedure to invoke this feature is:

- Enable `CONFIG_CODE_DATA_RELOCATION` in the `prj.conf` file
- Inside the `CMakeLists.txt` file in the project, mention all the files that need relocation.

  `zephyr_code_relocate(src/*.c SRAM2)`

  Where the first argument is the file/files and the second argument is the memory where it must be placed.

  Note: The file argument supports limited regular expressions. function zephyr_code_relocate() can be called as many times as required. This step has to be performed before the inclusion of boilerplate.cmake.

Additional Configurations

This section shows additional configuration options that can be set in `CMakeLists.txt`

- if the memory is SRAM1, SRAM2, CCD, or AON, then place the full object in the sections for example:

```
zephyr_code_relocate(src/file1.c SRAM2)
zephyr_code_relocate(src/file2.c.c SRAM)
```

- if the memory type is appended with _DATA, _TEXT, _RODATA or _BSS, only the selected memory is placed in the required memory region. for example:

```
zephyr_code_relocate(src/file1.c SRAM2_DATA)
zephyr_code_relocate(src/file2.c.c SRAM2_TEXT)
```

- Multiple regions can also be appended together such as: SRAM2_DATA_BSS. This will place data and bss inside SRAM2.

Sample

A sample showcasing this feature is provided at `$ZEPHYR_BASE/samples/application_development/code_relocation/`

This is an example of using the code relocation feature.

This example will place .text, .data, .bss from 3 files to various parts in the SRAM using a custom linker file derived from `include/arch/arm/aarch32/cortex_m/scripts/linker.ld`

8.7 Cryptography

The crypto section contains information regarding the cryptographic primitives supported by the Zephyr kernel. Use the information to understand the principles behind the operation of the different algorithms and how they were implemented.

The following crypto libraries have been included:

8.7.1 TinyCrypt Cryptographic Library

Overview

The TinyCrypt Library provides an implementation for targeting constrained devices with a minimal set of standard cryptography primitives, as listed below. To better serve applications targeting constrained devices, TinyCrypt implementations differ from the standard specifications (see the Important Remarks section for some important differences). Certain cryptographic primitives depend on other primitives, as mentioned in the list below.

Aside from the Important Remarks section below, valuable information on the usage, security and technicalities of each cryptographic primitive are found in the corresponding header file.

- SHA-256:
  - Type of primitive: Hash function.
  - Standard Specification: NIST FIPS PUB 180-4.
  - Requires: –
- HMAC-SHA256:
  - Type of primitive: Message authentication code.
  - Standard Specification: RFC 2104.
  - Requires: SHA-256
- HMAC-PRNG:
  - Type of primitive: Pseudo-random number generator.
  - Standard Specification: NIST SP 800-90A.
  - Requires: SHA-256 and HMAC-SHA256.
- AES-128:
  - Type of primitive: Block cipher.
  - Standard Specification: NIST FIPS PUB 197.
  - Requires: –
- AES-CBC mode:
  - Type of primitive: Encryption mode of operation.
  - Standard Specification: NIST SP 800-38A.
  - Requires: AES-128.
- AES-CTR mode:
  - Type of primitive: Encryption mode of operation.
  - Standard Specification: NIST SP 800-38A.
  - Requires: AES-128.
- AES-CMAC mode:
  - Type of primitive: Message authentication code.
  - Standard Specification: NIST SP 800-38B.
  - Requires: AES-128.
- AES-CCM mode:

  - Type of primitive: Authenticated encryption.
  - Standard Specification: NIST SP 800-38C.
  - Requires: AES-128.
- ECC-DH:
  - Type of primitive: Key exchange.
  - Standard Specification: RFC 6090.
  - Requires: ECC auxiliary functions (ecc.h/c).
- ECC-DSA:
  - Type of primitive: Digital signature.
  - Standard Specification: RFC 6090.
  - Requires: ECC auxiliary functions (ecc.h/c).

Design Goals

- Minimize the code size of each cryptographic primitive. This means minimize the size of a board-independent implementation, as presented in TinyCrypt. Note that various applications may require further features, optimizations with respect to other metrics and countermeasures for particular threats. These peculiarities would increase the code size and thus are not considered here.
- Minimize the dependencies among the cryptographic primitives. This means that it is unnecessary to build and allocate object code for more primitives than the ones strictly required by the intended application. In other words, one can select and compile only the primitives required by the application.

Important Remarks

The cryptographic implementations in TinyCrypt library have some limitations. Some of these limitations are inherent to the cryptographic primitives themselves, while others are specific to TinyCrypt. Some of these limitations are discussed in-depth below.

General Remarks

- TinyCrypt does **not** intend to be fully side-channel resistant. Due to the variety of side-channel attacks, many of them making certain boards vulnerable. In this sense, instead of penalizing all library users with side-channel countermeasures such as increasing the overall code size, TinyCrypt only implements certain generic timing-attack countermeasures.

Specific Remarks

- SHA-256:
  - The number of bits_hashed in the state is not checked for overflow. Note however that this will only be a problem if you intend to hash more than 2^64 bits, which is an extremely large window.
- HMAC:
  - The HMAC verification process is assumed to be performed by the application. This compares the computed tag with some given tag. Note that conventional memory-comparison methods (such as memcmp function) might be vulnerable to timing attacks; thus be sure to use a constant-time memory comparison function (such as compare_constant_time function provided in lib/utils.c).

- HMAC-PRNG:
  - Before using HMAC-PRNG, you *must* find an entropy source to produce a seed. PRNGs only stretch the seed into a seemingly random output of arbitrary length. The security of the output is exactly equal to the unpredictability of the seed.
  - NIST SP 800-90A requires three items as seed material in the initialization step: entropy seed, personalization and a nonce (which is not implemented). TinyCrypt requires the personalization byte array and automatically creates the entropy seed using a mandatory call to the re-seed function.
- AES-128:
  - The current implementation does not support other key-lengths (such as 256 bits). Note that if you need AES-256, it doesn't sound as though your application is running in a constrained environment. AES-256 requires keys twice the size as for AES-128, and the key schedule is 40% larger.
- CTR mode:
  - The AES-CTR mode limits the size of a data message they encrypt to 2^32 blocks. If you need to encrypt larger data sets, your application would need to replace the key after 2^32 block encryptions.
- CBC mode:
  - TinyCrypt CBC decryption assumes that the iv and the ciphertext are contiguous (as produced by TinyCrypt CBC encryption). This allows for a very efficient decryption algorithm that would not otherwise be possible.
- CMAC mode:
  - AES128-CMAC mode of operation offers 64 bits of security against collision attacks. Note however that an external attacker cannot generate the tags him/herself without knowing the MAC key. In this sense, to attack the collision property of AES128-CMAC, an external attacker would need the cooperation of the legal user to produce an exponentially high number of tags (e.g. 2^64) to finally be able to look for collisions and benefit from them. As an extra precaution, the current implementation allows to at most 2^48 calls to tc_cmac_update function before re-calling tc_cmac_setup (allowing a new key to be set), as suggested in Appendix B of SP 800-38B.
- CCM mode:
  - There are a few tradeoffs for the selection of the parameters of CCM mode. In special, there is a tradeoff between the maximum number of invocations of CCM under a given key and the maximum payload length for those invocations. Both things are related to the parameter 'q' of CCM mode. The maximum number of invocations of CCM under a given key is determined by the nonce size, which is: 15-q bytes. The maximum payload length for those invocations is defined as 2^(8q) bytes.

    To achieve minimal code size, TinyCrypt CCM implementation fixes q = 2, which is a quite reasonable choice for constrained applications. The implications of this choice are:

    The nonce size is: 13 bytes.

    The maximum payload length is: 2^16 bytes = 65 KB.

    The mac size parameter is an important parameter to estimate the security against collision attacks (that aim at finding different messages that produce the same authentication tag). TinyCrypt CCM implementation accepts any even integer between 4 and 16, as suggested in SP 800-38C.
  - TinyCrypt CCM implementation accepts associated data of any length between 0 and (2^16 - 2^8) = 65280 bytes.
  - TinyCrypt CCM implementation accepts:

    - Both non-empty payload and associated data (it encrypts and authenticates the payload and only authenticates the associated data);
    - Non-empty payload and empty associated data (it encrypts and authenticates the payload);
    - Non-empty associated data and empty payload (it degenerates to an authentication-only mode on the associated data).
  - RFC-3610, which also specifies CCM, presents a few relevant security suggestions, such as: it is recommended for most applications to use a mac size greater than 8. Besides, it is emphasized that the usage of the same nonce for two different messages which are encrypted with the same key obviously destroys the security properties of CCM mode.
- ECC-DH and ECC-DSA:
  - TinyCrypt ECC implementation is based on nano-ecc (see https://github.com/iSECPartners/nano-ecc) which in turn is based on micro-ecc (see https://github.com/kmackay/micro-ecc). In the original nano and micro-ecc documentation, there is an important remark about the way integers are represented:

    "Integer representation: To reduce code size, all large integers are represented using little-endian words - so the least significant word is first. You can use the 'ecc_bytes2native()' and 'ecc_native2bytes()' functions to convert between the native integer representation and the standardized octet representation."

Examples of Applications

It is possible to do useful cryptography with only the given small set of primitives. With this list of primitives it becomes feasible to support a range of cryptography usages:

- Measurement of code, data structures, and other digital artifacts (SHA256);
- Generate commitments (SHA256);
- Construct keys (HMAC-SHA256);
- Extract entropy from strings containing some randomness (HMAC-SHA256);
- Construct random mappings (HMAC-SHA256);
- Construct nonces and challenges (HMAC-PRNG);
- Authenticate using a shared secret (HMAC-SHA256);
- Create an authenticated, replay-protected session (HMAC-SHA256 + HMAC-PRNG);
- Authenticated encryption (AES-128 + AES-CCM);
- Key-exchange (EC-DH);
- Digital signature (EC-DSA);

Test Vectors

The library provides a test program for each cryptographic primitive (see 'test' folder). Besides illustrating how to use the primitives, these tests evaluate the correctness of the implementations by checking the results against well-known publicly validated test vectors.

For the case of the HMAC-PRNG, due to the necessity of performing an extensive battery test to produce meaningful conclusions, we suggest the user to evaluate the unpredictability of the implementation by using the NIST Statistical Test Suite (see References).

For the case of the EC-DH and EC-DSA implementations, most of the test vectors were obtained from the site of the NIST Cryptographic Algorithm Validation Program (CAVP), see References.

References

- NIST FIPS PUB 180-4 (SHA-256)
- NIST FIPS PUB 197 (AES-128)
- NIST SP800-90A (HMAC-PRNG)
- NIST SP 800-38A (AES-CBC and AES-CTR)
- NIST SP 800-38B (AES-CMAC)
- NIST SP 800-38C (AES-CCM)
- NIST Statistical Test Suite
- NIST Cryptographic Algorithm Validation Program (CAVP) site
- RFC 2104 (HMAC-SHA256)
- RFC 6090 (ECC-DH and ECC-DSA)

8.8 Flashing and Hardware Debugging

8.8.1 Flash & Debug Host Tools

This guide describes the software tools you can run on your host workstation to flash and debug Zephyr applications.

Zephyr's west tool has built-in support for all of these in its `flash`, `debug`, `debugserver`, and `attach` commands, provided your board hardware supports them and your Zephyr board directory's `board.cmake` file declares that support properly. See *Building, Flashing and Debugging* for more information on these commands.

SAM Boot Assistant (SAM-BA)

Atmel SAM Boot Assistant (Atmel SAM-BA) allows In-System Programming (ISP) from USB or UART host without any external programming interface. Zephyr allows users to develop and program boards with SAM-BA support using *west*. Zephyr supports devices with/without ROM bootloader and both extensions from Arduino and Adafruit. Full support was introduced in Zephyr SDK 0.12.0.

The typical command to flash the board is:

```
west flash [ -r bossac ] [ -p /dev/ttyX ]
```

Flash configuration for devices:

With ROM bootloader

These devices don't need any special configuration. After building your application, just run `west flash` to flash the board.

Without ROM bootloader

For these devices, the user should:

1. Define flash partitions required to accommodate the bootloader and application image; see *Flash map* for details.
2. Have board `.defconfig` file with the `CONFIG_USE_DT_CODE_PARTITION` Kconfig option set to `y` to instruct the build system to use these partitions for code relocation. This option can also be set in `prj.conf` or any other Kconfig fragment.
3. Build and flash the SAM-BA bootloader on the device.

With compatible SAM-BA bootloader

For these devices, the user should:

1. Define flash partitions required to accommodate the bootloader and application image; see *Flash map* for details.
2. Have board `.defconfig` file with the `CONFIG_BOOTLOADER_BOSSA` Kconfig option set to `y`. This will automatically select the `CONFIG_USE_DT_CODE_PARTITION` Kconfig option which instruct the build system to use these partitions for code relocation. The board `.defconfig` file should have `CONFIG_BOOTLOADER_BOSSA_ARDUINO` , `CONFIG_BOOTLOADER_BOSSA_ADAFRUIT_UF2` or the `CONFIG_BOOTLOADER_BOSSA_LEGACY` Kconfig option set to `y` to select the right compatible SAM-BA bootloader mode. These options can also be set in `prj.conf` or any other Kconfig fragment.
3. Build and flash the SAM-BA bootloader on the device.

Note: The `CONFIG_BOOTLOADER_BOSSA_LEGACY` Kconfig option should be used as last resource. Try configure first with Devices without ROM bootloader.

Typical flash layout and configuration For bootloaders that reside on flash, the devicetree partition layout is mandatory. For devices that have a ROM bootloader, they are mandatory when the application uses a storage or other non-application partition. In this special case, the boot partition should be omitted and code_partition should start from offset 0. It is necessary to define the partitions with sizes that avoid overlaps, always.

A typical flash layout for devices without a ROM bootloader is:

```
/ {
        chosen {
                zephyr,code-partition = &code_partition;
        };
};

&flash0 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                boot_partition: partition@0 {
                        label = "sam-ba";
                        reg = <0x00000000 0x2000>;
                        read-only;
                };

                code_partition: partition@2000 {
                        label = "code";
                        reg = <0x2000 0x3a000>;
                        read-only;
                };

                /*
                * The final 16 KiB is reserved for the application.
                * Storage partition will be used by FCB/LittleFS/NVS
                * if enabled.
                */
                storage_partition: partition@3c000 {
                        label = "storage";
```

(continues on next page)

(continued from previous page)

```
                        reg = <0x0003c000 0x00004000>;
                };
        };
};
```

A typical flash layout for devices with a ROM bootloader and storage partition is:

```
/ {
        chosen {
                zephyr,code-partition = &code_partition;
        };
};

&flash0 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                code_partition: partition@0 {
                        label = "code";
                        reg = <0x0 0xF0000>;
                        read-only;
                };

                /*
                * The final 64 KiB is reserved for the application.
                * Storage partition will be used by FCB/LittleFS/NVS
                * if enabled.
                */
                storage_partition: partition@F0000 {
                        label = "storage";
                        reg = <0x000F0000 0x00100000>;
                };
        };
};
```

Enabling SAM-BA runner In order to instruct Zephyr west tool to use the SAM-BA bootloader the `board.cmake` file must have `include(${ZEPHYR_BASE}/boards/common/bossac.board.cmake)` entry. Note that Zephyr tool accept more entries to define multiple runners. By default, the first one will be selected when using `west flash` command. The remaining options are available passing the runner option, for instance `west flash -r bossac`.

More implementation details can be found in the boards documentation. As a quick reference, see these three board documentation pages:

- sam4e_xpro (ROM bootloader)
- adafruit_feather_m0_basic_proto (Adafruit UF2 bootloader)
- arduino_nano_33_iot (Arduino bootloader)
- arduino_nano_33_ble (Arduino legacy bootloader)

J-Link Debug Host Tools

Segger provides a suite of debug host tools for Linux, macOS, and Windows operating systems:

- J-Link GDB Server: GDB remote debugging
- J-Link Commander: Command-line control and flash programming
- RTT Viewer: RTT terminal input and output
- SystemView: Real-time event visualization and recording

These debug host tools are compatible with the following debug probes:

- *LPC-Link2 J-Link Onboard Debug Probe*
- *OpenSDA J-Link Onboard Debug Probe*
- *J-Link External Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in J-Link Supported Devices.

Download and install the J-Link Software and Documentation Pack to get the J-Link GDB Server and Commander, and to install the associated USB device drivers. RTT Viewer and SystemView can be downloaded separately, but are not required.

Note that the J-Link GDB server does not yet support Zephyr RTOS-awareness.

OpenOCD Debug Host Tools

OpenOCD is a community open source project that provides GDB remote debugging and flash programming support for a wide range of SoCs. A fork that adds Zephyr RTOS-awareness is included in the Zephyr SDK; otherwise see Getting OpenOCD for options to download OpenOCD from official repositories.

These debug host tools are compatible with the following debug probes:

- *OpenSDA DAPLink Onboard Debug Probe*
- *J-Link External Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in OpenOCD Supported Devices.

Note: On Linux, openocd is available though the Zephyr SDK. Windows users should use the following steps to install openocd:

- Download openocd for Windows from here: OpenOCD Windows
- Copy bin and share dirs to `C:\Program Files\OpenOCD\`
- Add `C:\Program Files\OpenOCD\bin` to 'PATH' environment variable

---

pyOCD Debug Host Tools

pyOCD is an open source project from Arm that provides GDB remote debugging and flash programming support for Arm Cortex-M SoCs. It is distributed on PyPi and installed when you complete the *Get Zephyr and install Python dependencies* step in the Getting Started Guide. pyOCD includes support for Zephyr RTOS-awareness.

These debug host tools are compatible with the following debug probes:

- *OpenSDA DAPLink Onboard Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in pyOCD Supported Devices.

8.8.2 Debug Probes

A *debug probe* is special hardware which allows you to control execution of a Zephyr application running on a separate board. Debug probes usually allow reading and writing registers and memory, and support breakpoint debugging of the Zephyr application on your host workstation using tools like GDB. They may also support other debug software and more advanced features such as *tracing program execution*. For details on the related host software supported by Zephyr, see *Flash & Debug Host Tools*.

Debug probes are usually connected to your host workstation via USB; they are sometimes also accessible via an IP network or other means. They usually connect to the device running Zephyr using the JTAG or SWD protocols. Debug probes are either separate hardware devices or circuitry integrated into the same board which runs Zephyr.

Many supported boards in Zephyr include a second microcontroller that serves as an onboard debug probe, usb-to-serial adapter, and sometimes a drag-and-drop flash programmer. This eliminates the need to purchase an external debug probe and provides a variety of debug host tool options.

Several hardware vendors have their own branded onboard debug probe implementations: NXP LPC boards have *LPC-Link2*, NXP Kinetis (former Freescale) boards have *OpenSDA*, and ST boards have *ST-LINK*. Each onboard debug probe microcontroller can support one or more types of firmware that communicate with their respective debug host tools. For example, an OpenSDA microcontroller can be programmed with DAPLink firmware to communicate with pyOCD or OpenOCD debug host tools, or with J-Link firmware to communicate with J-Link debug host tools.

Debug Probes & Host Tools Compatibility Chart		Host Tools		
		J-Link Debug	**OpenOCD**	**pyOCD**
Debug Probes	**LPC-Link2 J-Link**	✓		
	OpenSDA DAPLink		✓	✓
	OpenSDA J-Link	✓		
	J-Link External	✓	✓	
	ST-LINK/V2-1	✓	✓	*some STM32 boards*

Some supported boards in Zephyr do not include an onboard debug probe and therefore require an external debug probe. In addition, boards that do include an onboard debug probe often also have an SWD or JTAG header to enable the use of an external debug probe instead. One reason this may be useful is that the onboard debug probe may have limitations, such as lack of support for advanced debuggers or high-speed tracing. You may need to adjust jumpers to prevent the onboard debug probe from interfering with the external debug probe.

LPC-Link2 J-Link Onboard Debug Probe

The LPC-Link2 J-Link is an onboard debug probe and usb-to-serial adapter supported on many NXP LPC and i.MX RT development boards.

This debug probe is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

This probe is realized by programming the LPC-Link2 microcontroller with J-Link LPC-Link2 firmware. Download and install LPCScrypt to get the firmware and programming scripts.

Note: Verify the firmware supports your board by visiting Firmware for LPCXpresso

1. Put the LPC-Link2 microcontroller into DFU boot mode by attaching the DFU jumper, then powering up the board.

2. Run the `program_JLINK` script.
3. Remove the DFU jumper and power cycle the board.

OpenSDA DAPLink Onboard Debug Probe

The OpenSDA DAPLink is an onboard debug probe and usb-to-serial adapter supported on many NXP Kinetis and i.MX RT development boards. It also includes drag-and-drop flash programming support.

This debug probe is compatible with the following debug host tools:

- *pyOCD Debug Host Tools*
- *OpenOCD Debug Host Tools*

This probe is realized by programming the OpenSDA microcontroller with DAPLink OpenSDA firmware. NXP provides OpenSDA DAPLink Board-Specific Firmwares.

Install the debug host tools before you program the firmware.

As with all OpenSDA debug probes, the steps for programming the firmware are:

1. Put the OpenSDA microcontroller into bootloader mode by holding the reset button while you power on the board. Note that "bootloader mode" in this context applies to the OpenSDA microcontroller itself, not the target microcontroller of your Zephyr application.
2. After you power on the board, release the reset button. A USB mass storage device called **BOOTLOADER** or **MAINTENANCE** will enumerate.
3. Copy the OpenSDA firmware binary to the USB mass storage device.
4. Power cycle the board, this time without holding the reset button. You should see three USB devices enumerate: a CDC device (serial port), a HID device (debug port), and a mass storage device (drag-and-drop flash programming).

OpenSDA J-Link Onboard Debug Probe

The OpenSDA J-Link is an onboard debug probe and usb-to-serial adapter supported on many NXP Kinetis and i.MX RT development boards.

This debug probe is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

This probe is realized by programming the OpenSDA microcontroller with J-Link OpenSDA firmware. Segger provides OpenSDA J-Link Generic Firmwares and OpenSDA J-Link Board-Specific Firmwares, where the latter is generally recommended when available. Board-specific firmwares are required for i.MX RT boards to support their external flash memories, whereas generic firmwares are compatible with all Kinetis boards.

Install the debug host tools before you program the firmware.

As with all OpenSDA debug probes, the steps for programming the firmware are:

1. Put the OpenSDA microcontroller into bootloader mode by holding the reset button while you power on the board. Note that "bootloader mode" in this context applies to the OpenSDA microcontroller itself, not the target microcontroller of your Zephyr application.
2. After you power on the board, release the reset button. A USB mass storage device called **BOOTLOADER** or **MAINTENANCE** will enumerate.
3. Copy the OpenSDA firmware binary to the USB mass storage device.
4. Power cycle the board, this time without holding the reset button. You should see two USB devices enumerate: a CDC device (serial port) and a vendor-specific device (debug port).

J-Link External Debug Probe

Segger J-Link is a family of external debug probes, including J-Link EDU, J-Link PLUS, J-Link ULTRA+, and J-Link PRO, that support a large number of devices from different hardware architectures and vendors.

This debug probe is compatible with the following debug host tools:

- *J-Link Debug Host Tools*
- *OpenOCD Debug Host Tools*

Install the debug host tools before you program the firmware.

ST-LINK/V2-1 Onboard Debug Probe

ST-LINK/V2-1 is a serial and debug adapter built into all Nucleo and Discovery boards. It provides a bridge between your computer (or other USB host) and the embedded target processor, which can be used for debugging, flash programming, and serial communication, all over a simple USB cable.

It is compatible with the following host debug tools:

- *OpenOCD Debug Host Tools*
- *J-Link Debug Host Tools*

For some STM32 based boards, it is also compatible with:

- *pyOCD Debug Host Tools*

While it works out of the box with OpenOCD, it requires some flashing to work with J-Link. To do this, SEGGER offers a firmware upgrading the ST-LINK/V2-1 on board on the Nucleo and Discovery boards. This firmware makes the ST-LINK/V2-1 compatible with J-LinkOB, allowing users to take advantage of most J-Link features like the ultra fast flash download and debugging speed or the free-to-use GDBServer.

More informations about upgrading ST-LINK/V2-1 to JLink or restore ST-Link/V2-1 firmware please visit: Segger over ST-Link

Flash and debug with ST-Link Using OpenOCD

OpenOCD is available by default on ST-Link and configured as the default flash and debug tool. Flash and debug can be done as follows:

```
# From the root of the zephyr repository
west build -b None samples/hello_world
west flash
```

```
# From the root of the zephyr repository
west build -b None samples/hello_world
west debug
```

Using Segger J-Link

Once STLink is flashed with SEGGER FW and J-Link GDB server is installed on your host computer, you can flash and debug as follows:

Use CMake with `-DBOARD_FLASH_RUNNER=jlink` to change the default OpenOCD runner to J-Link. Alternatively, you might add the following line to your application `CMakeList.txt` file.

```
set(BOARD_FLASH_RUNNER jlink)
```

If you use West (Zephyr's meta-tool) you can modify the default runner using the `--runner` (or `-r`) option.

```
west flash --runner jlink
```

To attach a debugger to your board and open up a debug console with `jlink`.

```
west debug --runner jlink
```

For more information about West and available options, see *West (Zephyr's meta-tool)*.

If you configured your Zephyr application to use Segger RTT console instead, open telnet:

```
$ telnet localhost 19021
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SEGGER J-Link V6.30f - Real time terminal output
J-Link STLink V21 compiled Jun 26 2017 10:35:16 V1.0, SN=773895351
Process: JLinkGDBServerCLExe
Zephyr Shell, Zephyr version: 1.12.99
Type 'help' for a list of available commands
shell>
```

If you get no RTT output you might need to disable other consoles which conflict with the RTT one if they are enabled by default in the particular sample or application you are running, such as disable UART_CONSOLE in menucon

Updating or restoring ST-Link firmware ST-Link firmware can be updated using STM32CubeProgrammer Tool. It is usually useful when facing flashing issues, for instance when using twister's device-testing option.

Once installed, you can update attached board ST-Link firmware with the following command

```
s java -jar ~/STMicroelectronics/STM32Cube/STM32CubeProgrammer/Drivers/
↪FirmwareUpgrade/STLinkUpgrade.jar -sn <board_uid>
```

Where board_uid can be obtained using twister's generate-hardware-map option. For more information about twister and available options, see *Test Runner (Twister)*.

8.9 Debugging and Tracing

8.9.1 Thread analyzer

The thread analyzer module enables all the Zephyr options required to track the thread information, e.g. thread stack size usage and other runtime thread runtime statistics.

The analysis is performed on demand when the application calls *thread_analyzer_run()* or *thread_analyzer_print()*.

For example, to build the synchronization sample with Thread Analyser enabled, do the following:

```
west build -b qemu_x86 samples/synchronization/ -- -DCONFIG_QEMU_ICOUNT=n -
↪DCONFIG_THREAD_ANALYZER=y \
-DCONFIG_THREAD_ANALYZER_USE_PRINTK=y -DCONFIG_THREAD_ANALYZER_AUTO=y \
-DCONFIG_THREAD_ANALYZER_AUTO_INTERVAL=5
```

When you run the generated application in Qemu, you will get the additional information from Thread Analyzer:

```
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 740 usage 284 / 1024 (27 %); CPU: 0 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 98 %
 idle 00             : STACK: unused 204 usage 116 / 320 (36 %); CPU: 0 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 7 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 9 %
 idle 00             : STACK: unused 204 usage 116 / 320 (36 %); CPU: 82 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 7 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 8 %
 idle 00             : STACK: unused 204 usage 116 / 320 (36 %); CPU: 83 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
```

Configuration

Configure this module using the following options.

- `THREAD_ANALYZER`: enable the module.
- `THREAD_ANALYZER_USE_PRINTK`: use printk for thread statistics.
- `THREAD_ANALYZER_USE_LOG`: use the logger for thread statistics.
- `THREAD_ANALYZER_AUTO`: run the thread analyzer automatically. You do not need to add any code to the application when using this option.
- `THREAD_ANALYZER_AUTO_INTERVAL`: the time for which the module sleeps between consecutive printing of thread analysis in automatic mode.
- `THREAD_ANALYZER_AUTO_STACK_SIZE`: the stack for thread analyzer automatic thread.
- `THREAD_NAME`: enable this option in the kernel to print the name of the thread instead of its ID.
- `THREAD_RUNTIME_STATS`: enable this option to print thread runtime data such as utilization (This options is automatically selected by THREAD_ANALYZER).

API documentation

group `thread_analyzer`

Module for analyzing threads.

This module implements functions and the configuration that simplifies thread analysis.

Typedefs

typedef void (*`thread_analyzer_cb`)(struct *thread_analyzer_info* *info)

Thread analyzer stack size callback function.

Callback function with thread analysis information.

Param info Thread analysis information.

Functions

void `thread_analyzer_run`(*thread_analyzer_cb* cb)

Run the thread analyzer and provide information to the callback.

This function analyzes the current state for all threads and calls a given callback on every thread found.

Parameters

- `cb` – The callback function handler

void `thread_analyzer_print`(void)

Run the thread analyzer and print stack size statistics.

This function runs the thread analyzer and prints the output in standard form.

struct `thread_analyzer_info`

#include <thread_analyzer.h>

Public Members

const char *`name`

The name of the thread or stringified address of the thread handle if name is not set.

size_t `stack_size`

The total size of the stack

size_t `stack_used`

Stack size in used

8.9.2 Core Dump

The core dump module enables dumping the CPU registers and memory content for offline debugging. This module is called when fatal error is encountered, and the data is printed or stored according to which backends are enabled.

Configuration

Configure this module using the following options.

- `DEBUG_COREDUMP`: enable the module.

Here are the options to enable output backends for core dump:

- `DEBUG_COREDUMP_BACKEND_LOGGING`: use log module for core dump output.
- `DEBUG_COREDUMP_BACKEND_NULL`: fallback core dump backend if other backends cannot be enabled. All output is sent to null.

Here are the choices regarding memory dump:

- `DEBUG_COREDUMP_MEMORY_DUMP_MIN`: only dumps the stack of the exception thread, its thread struct, and some other bare minimal data to support walking the stack in debugger. Use this only if absolute minimum of data dump is desired.

Usage

When the core dump module is enabled, during fatal error, CPU registers and memory content are being printed or stored according to which backends are enabled. This core dump data can fed into a custom made GDB server as a remote target for GDB (and other GDB compatible debuggers). CPU registers, memory content and stack can be examined in the debugger.

This usually involves the following steps:

1. Get the core dump log from the device depending on enabled backends. For example, if the log module backend is used, get the log output from the log module backend.
2. Convert the core dump log into a binary format that can be parsed by the GDB server. For example, scripts/coredump/coredump_serial_log_parser.py can be used to convert the serial console log into a binary file.
3. Start the custom GDB server using the script scripts/coredump/coredump_gdbserver.py with the core dump binary log file, and the Zephyr ELF file as parameters.
4. Start the debugger corresponding to the target architecture.

Example This example uses the log module backend tied to serial console. This was done on `qemu_x86` where a null pointer was dereferenced.

This is the core dump log from the serial console, and is stored in `coredump.log`:

```
Booting from ROM..*** Booting Zephyr OS build zephyr-v2.3.0-1840-g7bba91944a63  ***
Hello World! qemu_x86
E: Page fault at address 0x0 (error code 0x2)
E: Linear address not present in page tables
E:   PDE: 0x0000000000115827 Writable, User, Execute Enabled
E:   PTE: Non-present
E: EAX: 0x00000000, EBX: 0x00000000, ECX: 0x00119d74, EDX: 0x000003f8
E: ESI: 0x00000000, EDI: 0x00101aa7, EBP: 0x00119d10, ESP: 0x00119d00
E: EFLAGS: 0x00000206 CS: 0x0008 CR3: 0x00119000
E: call trace:
E: EIP: 0x00100459
E:      0x00100477 (0x0)
E:      0x00100492 (0x0)
E:      0x001004c8 (0x0)
E:      0x00105465 (0x105465)
E:      0x00101abe (0x0)
E: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
```

(continues on next page)

(continued from previous page)

```
E: Current thread: 0x00119080 (unknown)
E: #CD:BEGIN#
E: #CD:5a4501000100050000000000
E: #CD:4101003800
E: #CD:0e0000002000000000000000749d1100f80300000000000009d1100109d1100
E: #CD:00000000a71a1000590410000602000008000000000901100
E: #CD:4d010080901100e0901100
E: #CD:010000000000000000000000018000000000000000000000000000000000000000
E: #CD:000000000000000000000000000000e364100000000000000000004c9c1100
E: #CD:00000000000000000000000b4991100000400000000000fc03000000000000
E: #CD:4d0100b4991100b49d1100
E: #CD:f8030000020000000200000002000000f8030000fd03000a02000000dc9e1100
E: #CD:149a1160fd03000002000000dc9e1100249a110087201000049f11000a000000
E: #CD:349a11000a4f1000049f11000a9e1100449a11000a8b1000020000000200000000
E: #CD:449a1100388b1000049f11000a000000549a1100ad201000049f11000a000000
E: #CD:749a11000a201000049f11000a000000649a11000a201000049f11000a000000
E: #CD:749a1100e8201000049f11000a000000949a1100890b10000a0000000a000000
E: #CD:a49a1100890b10000a0000000a000000f8030000189b1100020000000200000000
E: #CD:f49a1100289b11000a000000189b1100049b11009b0710000a000000289b1100
E: #CD:f49a110087201000049f110045000000f49a1100509011000a00000020901100
E: #CD:f49a110060901100049f1100ffffffff00000000000000000049f1100ffffffff
E: #CD:000000000000000630b1000189b1100349b1100af0b1000630b1000289b1100
E: #CD:55891000789b1100000000002090110054 9b1100480000004a891000609b1100
E: #CD:649b1100d00b10004a891000709b110000000000609b11000a00000000000000
E: #CD:849b1100709b11004a8910000000000949b1100794a1000000000058901100
E: #CD:20901100c34a10000a00001734020000d001000000000000d49b110038000000
E: #CD:c49b110078481000b4991100000400000000000000000000c9c11000c9c1100
E: #CD:149c110000000000d49b110038000000f49b1100da481000b499110000040000
E: #CD:0e0000002000000000000000744d0100b4991100b49d1100009d1100109d1100
E: #CD:149c110099471000b4991100000400000800000000901100ad861000409c1100
E: #CD:349c1100e947100080901100000000000349c1100b64710008086100045000000
E: #CD:849c11002d53100000000000d09c1100809011002086 1000f5ffffff8c9c1100
E: #CD:00000000000000000000000a71a1000a49c1100020200008090110000000000
E: #CD:a49c1100020200008000000000000000a49c1100193710000000000d09c1100
E: #CD:0c9d0000bc9c0000b49d1100b4991100c49c1100ae37100000000000d09c1100
E: #CD:0800000000000000c8881000000000000109d11005d031000d09c1100009d1100
E: #CD:109d1100000000000000000000a71a1000f80300000000000749d110002000000
E: #CD:5904100008000000060200000e0000000202000002020000000000002c9d1100
E: #CD:7704100000000000d00b1000c9881000549d1100000000000489d110092041000
E: #CD:00000000689d1100549d11000000000000000000000689d1100c804100000000000
E: #CD:c0881000000000007c9d110000000000749d11007c9d1100655410006554 1000
E: #CD:0000000000000009c9d1100be1a1000000000000000000000000000000038041000
E: #CD:0800000002020000000000000000000f45310000000000000000000000000000
E: #CD:END#
E: Halting system
```

1. Run the core dump serial log converter:

   ```
   ./scripts/coredump/coredump_serial_log_parser.py coredump.log coredump.bin
   ```

2. Start the custom GDB server:

   ```
   ./scripts/coredump/coredump_gdbserver.py build/zephyr/zephyr.elf coredump.bin
   ```

3. Start GDB:

   ```
   <path to SDK>/x86_64-zephyr-elf/bin/x86_64-zephyr-elf-gdb build/zephyr/zephyr.elf
   ```

4. Inside GDB, connect to the GDB server via port 1234:

```
(gdb) target remote localhost:1234
```

5. Examine the CPU registers:

```
(gdb) info registers
```

Output from GDB:

```
eax            0x0                 0
ecx            0x119d74            1154420
edx            0x3f8               1016
ebx            0x0                 0
esp            0x119d00            0x119d00 <z_main_stack+844>
ebp            0x119d10            0x119d10 <z_main_stack+860>
esi            0x0                 0
edi            0x101aa7            1055399
eip            0x100459            0x100459 <func_3+16>
eflags         0x206               [ PF IF ]
cs             0x8                 8
ss             <unavailable>
ds             <unavailable>
es             <unavailable>
fs             <unavailable>
gs             <unavailable>
```

6. Examine the backtrace:

```
(gdb) bt
```

Output from GDB:

```
#0  0x00100459 in func_3 (addr=0x0) at zephyr/rtos/zephyr/samples/hello_world/
↪src/main.c:14
#1  0x00100477 in func_2 (addr=0x0) at zephyr/rtos/zephyr/samples/hello_world/
↪src/main.c:21
#2  0x00100492 in func_1 (addr=0x0) at zephyr/rtos/zephyr/samples/hello_world/
↪src/main.c:28
#3  0x001004c8 in main () at zephyr/rtos/zephyr/samples/hello_world/src/main.c:42
```

File Format

The core dump binary file consists of one file header, one architecture-specific block, and multiple memory blocks. All numbers in the headers below are little endian.

File Header The file header consists of the following fields:

Table 4: Core dump binary file header

Field	Data Type	Description
ID	`char[2]`	`Z`, `E` as identifier of file.
Header version	`uint16_t`	Identify the version of the header. This needs to be incremented whenever the header struct is modified. This allows parser to reject older header versions so it will not incorrectly parse the header.
Target code	`uint16_t`	Indicate which target (e.g. architecture or SoC) so the parser can instantiate the correct register block parser.
Pointer size	'uint8_t'	Size of `uintptr_t` in power of 2. (e.g. 5 for 32-bit, 6 for 64-bit). This is needed to accommodate 32-bit and 64-bit target in parsing the memory block addresses.
Flags	`uint8_t`	
Fatal error reason	`unsigned int`	Reason for the fatal error, as the same in `enum k_fatal_error_reason` defined in include/fatal.h

Architecture-specific Block The architecture-specific block contains the byte stream of data specific to the target architecture (e.g. CPU registers)

Table 5: Architecture-specific Block

Field	Data Type	Description
ID	`char`	`A` to indiciate this is a architecture-specific block.
Header version	`uint16_t`	Identify the version of this block. To be interpreted by the target architecture specific block parser.
Number of bytes	`uint16_t`	Number of bytes following the header which contains the byte stream for target data. The format of the byte stream is specific to the target and is only being parsed by the target parser.
Register byte stream	`uint8_t[]`	Contains target architecture specific data.

Memory Block The memory block contains the start and end addresses and the data within the memory region.

Table 6: Memory Block

Field	Data Type	Description
ID	`char`	`M` to indiciate this is a memory block.
Header version	`uint16_t`	Identify the version of the header. This needs to be incremented whenever the header struct is modified. This allows parser to reject older header versions so it will not incorrectly parse the header.
Start address	`uintptr_t`	The start address of the memory region.
End address	`uintptr_t`	The end address of the memory region.
Memory byte stream	`uint8_t[]`	Contains the memory content between the start and end addresses.

Adding New Target

The architecture-specific block is target specific and requires new dumping routine and parser for new targets. To add a new target, the following needs to be done:

1. Add a new target code to the `enum coredump_tgt_code` in include/debug/coredump.h.

2. Implement *arch_coredump_tgt_code_get()* simply to return the newly introducted target code.
3. Implement *arch_coredump_info_dump()* to construct a target architecture block and call *coredump_buffer_output()* to output the block to core dump backend.
4. Add a parser to the core dump GDB stub scripts under `scripts/coredump/gdbstubs/`
    1. Extends the `gdbstubs.gdbstub.GdbStub` class.
    2. During `__init__`, store the GDB signal corresponding to the exception reason in `self.gdb_signal`.
    3. Parse the architecture-specific block from `self.logfile.get_arch_data()`. This needs to match the format as implemented in step 3 (inside *arch_coredump_info_dump()*).
    4. Implement the abstract method `handle_register_group_read_packet` where it returns the register group as GDB expected. Refer to GDB's code and documentation on what it is expecting for the new target.
    5. Optionally implement `handle_register_single_read_packet` for registers not covered in the g packet.
5. Extend `get_gdbstub()` in scripts/coredump/gdbstubs/__init__.py to return the newly implemented GDB stub.

API documentation

group `coredump_apis`

Coredump APIs.

Functions

void `coredump`(unsigned int reason, const z_arch_esf_t *esf, struct *k_thread* *thread)

Perform coredump.

Normally, this is called inside z_fatal_error() to generate coredump when a fatal error is encountered. This can also be called on demand whenever a coredump is desired.

Parameters

- `reason` – Reason for the fatal error
- `esf` – Exception context
- `thread` – Thread information to dump

void `coredump_memory_dump`(uintptr_t start_addr, uintptr_t end_addr)

Dump memory region.

Parameters

- `start_addr` – Start address of memory region to be dumped
- `end_addr` – End address of memory region to be dumped

void `coredump_buffer_output`(uint8_t *buf, size_t buflen)

Output the buffer via coredump.

This outputs the buffer of byte array to the coredump backend. For example, this can be called to output the coredump section containing registers, or a section for memory dump.

Parameters

- `buf` – Buffer to be send to coredump output
- `buflen` – Buffer length

int coredump_query(enum coredump_query_id query_id, void *arg)

Perform query on coredump subsystem.

Query the coredump subsystem for information, for example, if there is an error.

Parameters

- query_id – **[in]** Query ID
- arg – **[inout]** Pointer to argument for exchanging information

Returns Depends on the query

int coredump_cmd(enum coredump_cmd_id query_id, void *arg)

Perform command on coredump subsystem.

Perform certain on coredump subsystem, for example, output the stored coredump via logging.

Parameters

- cmd_id – **[in]** Command ID
- arg – **[inout]** Pointer to argument for exchanging information

Returns Depends on the command

group arch-coredump

Functions

void arch_coredump_info_dump(const z_arch_esf_t *esf)

Architecture-specific handling during coredump.

This dumps architecture-specific information during coredump.

Parameters

- esf – Exception Stack Frame (arch-specific)

uint16_t arch_coredump_tgt_code_get(void)

Get the target code specified by the architecture.

8.9.3 GDB stub

Overview

The gdbstub feature provides an implementation of the GDB Remote Serial Protocol (RSP) that allows you to remotely debug Zephyr using GDB.

The protocol supports different connection types: serial, UDP/IP and TCP/IP. Zephyr currently supports only serial device communication.

The GDB program acts as the client while Zephyr acts as the server. When this feature is enabled, Zephyr stops its execution after gdb_init() starts gdbstub service and waits for a GDB connection. Once a connection is established it is possible to synchronously interact with Zephyr. Note that currently it is not possible to asynchronously send commands to the target.

Enable this feature with the CONFIG_GDBSTUB option.

Features

The following features are supported:

- Add and remove breakpoints
- Continue and step the target
- Print backtrace
- Read or write general registers
- Read or write the memory

8.9.4 Tracing

Overview

The tracing feature provides hooks that permits you to collect data from your application and allows tools running on a host to visualize the inner-working of the kernel and various subsystems.

Every system has application-specific events to trace out. Historically, that has implied:

1. Determining the application-specific payload,
2. Choosing suitable serialization-format,
3. Writing the on-target serialization code,
4. Deciding on and writing the I/O transport mechanics,
5. Writing the PC-side deserializer/parser,
6. Writing custom ad-hoc tools for filtering and presentation.

An application can use one of the existing formats or define a custom format by overriding the macros declared in include/tracing/tracing.h.

Different formats, transports and host tools are avialable and supported in Zephyr.

In fact, I/O varies greatly from system to system. Therefore, it is instructive to create a taxonomy for I/O types when we must ensure the interface between payload/format (Top Layer) and the transport mechanics (bottom Layer) is generic and efficient enough to model these. See the *I/O taxonomy* section below.

Serialization Formats

Common Trace Format (CTF) Support Common Trace Format, CTF, is an open format and language to describe trace formats. This enables tool reuse, of which line-textual (babeltrace) and graphical (TraceCompass) variants already exist.

CTF should look familiar to C programmers but adds stronger typing. See CTF - A Flexible, High-performance Binary Trace Format.

CTF allows us to formally describe application specific payload and the serialization format, which enables common infrastructure for host tools and parsers and tools for filtering and presentation.

A Generic Interface In CTF, an event is serialized to a packet containing one or more fields. As seen from *I/O taxonomy* section below, a bottom layer may:

- perform actions at transaction-start (e.g. mutex-lock),
- process each field in some way (e.g. sync-push emit, concat, enqueue to thread-bound FIFO),
- perform actions at transaction-stop (e.g. mutex-release, emit of concat buffer).

CTF Top-Layer Example The CTF_EVENT macro will serialize each argument to a field:

```
/* Example for illustration */
static inline void ctf_top_foo(uint32_t thread_id, ctf_bounded_string_t name)
{
  CTF_EVENT(
    CTF_LITERAL(uint8_t, 42),
    thread_id,
    name,
    "hello, I was emitted from function: ",
    __func__  /* __func__ is standard since C99 */
  );
}
```

How to serialize and emit fields as well as handling alignment, can be done internally and statically at compile-time in the bottom-layer.

The CTF top layer is enabled using the configuration option `CONFIG_TRACING_CTF` and can be used with the different transport backends both in synchronous and asynchronous modes.

SEGGER SystemView Support Zephyr provides built-in support for SEGGER SystemView that can be enabled in any application for platforms that have the required hardware support.

The payload and format used with SystemView is custom to the application and relies on RTT as a transport. Newer versions of SystemView support other transports, for example UART or using snapshot mode (both still not supported in Zephyr).

To enable tracing support with SEGGER SystemView add the configuration option `CONFIG_SEGGER_SYSTEMVIEW` to your project configuration file and set it to *y*. For example, this can be added to the synchronization_sample to visualize fast switching between threads. SystemView can also be used for post-mortem tracing, which can be enabled with `CONFIG_SEGGER_SYSVIEW_POST_MORTEM_MODE`. In this mode, a debugger can be attached after the system has crashed using `west attach` after which the latest data from the internal RAM buffer can be loaded into SystemView:

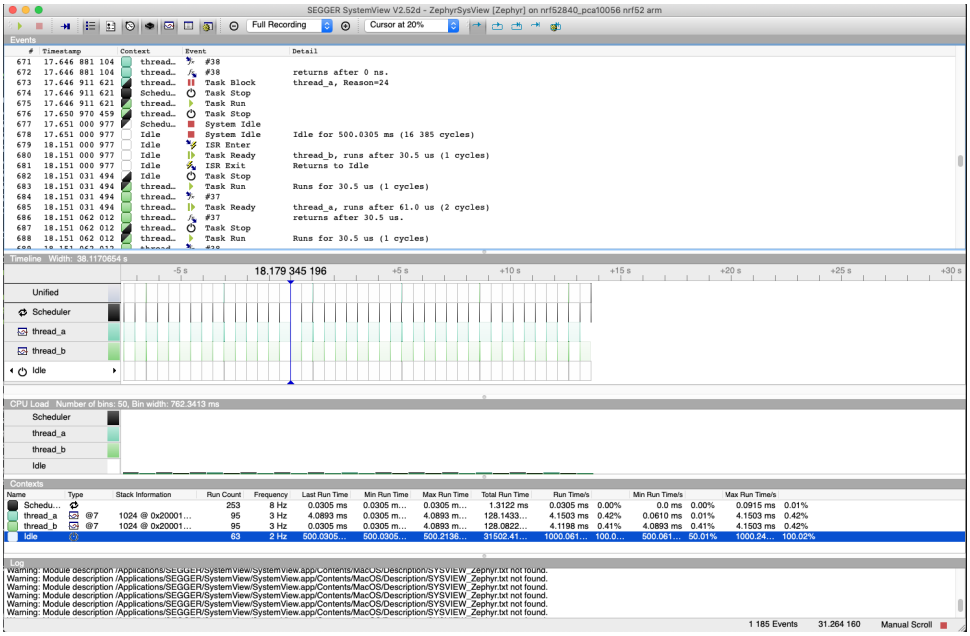
```
CONFIG_STDOUT_CONSOLE=y
# enable to use thread names
CONFIG_THREAD_NAME=y
CONFIG_SEGGER_SYSTEMVIEW=y
CONFIG_USE_SEGGER_RTT=y
CONFIG_TRACING=y
# enable for post-mortem tracing
CONFIG_SEGGER_SYSVIEW_POST_MORTEM_MODE=n
```

Recent versions of SEGGER SystemView come with an API translation table for Zephyr which is incomplete and does not match the current level of support available in Zephyr. To use the latest Zephyr API description table, copy the file available in the tree to your local configuration directory to override the builtin table:

```
# On Linux and MacOS
cp ZEPHYR_BASE/subsys/tracing/sysview/SYSVIEW_Zephyr.txt ~/.config/SEGGER/
```

User-Defined Tracing This tracing format allows the user to define functions to perform any work desired when a task is switched in or out, when an interrupt is entered or exited, and when the cpu is idle.

Examples include: - simple toggling of GPIO for external scope tracing while minimizing extra cpu load - generating/outputting trace data in a non-standard or proprietary format that can not be supported by the other tracing systems



The following functions can be defined by the user: - `void sys_trace_thread_switched_in_user(struct k_thread *thread)` - `void sys_trace_thread_switched_out_user(struct k_thread *thread)` - `void sys_trace_isr_enter_user()` - `void sys_trace_isr_exit_user()` - `void sys_trace_idle_user()`

Enable this format with the `CONFIG_TRACING_USER` option.

Transport Backends

The following backends are currently supported:

- UART
- USB
- File (Using native posix port)
- RTT (With SystemView)
- RAM (buffer to be retrieved by a debugger)

Using Tracing

The sample samples/subsys/tracing demonstrates tracing with different formats and backends.

To get started, the simplest way is to use the CTF format with the `native_posix` port, build the sample as follows:

Using west:

```
west build -b native_posix samples/subsys/tracing -- -DCONF_FILE=prj_native_posix_ctf.
↪conf
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -B build -GNinja -DBOARD=native_posix -DCONF_FILE=prj_native_posix_ctf.conf␣
↪samples/subsys/tracing
```

(continues on next page)

(continued from previous page)

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

You can then run the resulting binary with the option `-trace-file` to generate the tracing data:

```
mkdir data
cp $ZEPHYR_BASE/subsys/tracing/ctf/tsdl/metadata data/
./build/zephyr/zephyr.exe -trace-file=data/channel0_0
```

The resulting CTF output can be visualized using babeltrace or TraceCompass by pointing the tool to the `data` directory with the metadata and trace files.

Using RAM backend For devices that do not have available I/O for tracing such as USB or UART but have enough RAM to collect trace datas, the ram backend can be enabled with configuration `CONFIG_TRACING_BACKEND_RAM`. Adjust `CONFIG_RAM_TRACING_BUFFER_SIZE` to be able to record enough traces for your needs. Then thanks to a runtime debugger such as gdb this buffer can be fetched from the target to an host computer:

```
(gdb) dump binary memory data/channel0_0 <ram_tracing_start> <ram_tracing_end>
```

The resulting channel0_0 file have to be placed in a directory with the `metadata` file like the other backend.

Visualisation Tools

TraceCompass TraceCompass is an open source tool that visualizes CTF events such as thread scheduling and interrupts, and is helpful to find unintended interactions and resource conflicts on complex systems.

See also the presentation by Ericsson, Advanced Trouble-shooting Of Real-time Systems.

Future LTTng Inspiration

Currently, the top-layer provided here is quite simple and bare-bones, and needlessly copied from Zephyr's Segger SystemView debug module.

For an OS like Zephyr, it would make sense to draw inspiration from Linux's LTTng and change the top-layer to serialize to the same format. Doing this would enable direct reuse of TraceCompass' canned analyses for Linux. Alternatively, LTTng-analyses in TraceCompass could be customized to Zephyr. It is ongoing work to enable TraceCompass visibility of Zephyr in a target-agnostic and open source way.

I/O Taxonomy

- Atomic Push/Produce/Write/Enqueue:
  - **synchronous:** means data-transmission has completed with the return of the call.
  - **asynchronous:** means data-transmission is pending or ongoing with the return of the call. Usually, interrupts/callbacks/signals or polling is used to determine completion.
  - **buffered:** means data-transmissions are copied and grouped together to form a larger ones. Usually for amortizing overhead (burst dequeue) or jitter-mitigation (steady dequeue).

  Examples:

  - **sync unbuffered** E.g. PIO via GPIOs having steady stream, no extra FIFO memory needed. Low jitter but may be less efficient (cant amortize the overhead of writing).

  - **sync buffered** E.g. `fwrite()` or enqueuing into FIFO. Blockingly burst the FIFO when its buffer-waterlevel exceeds threshold. Jitter due to bursts may lead to missed deadlines.
  - **async unbuffered** E.g. DMA, or zero-copying in shared memory. Be careful of data hazards, race conditions, etc!
  - **async buffered** E.g. enqueuing into FIFO.

- Atomic Pull/Consume/Read/Dequeue:
  - **synchronous:** means data-reception has completed with the return of the call.
  - **asynchronous:** means data-reception is pending or ongoing with the return of the call. Usually, interrupts/callbacks/signals or polling is used to determine completion.
  - **buffered:** means data is copied-in in larger chunks than request-size. Usually for amortizing wait-time.

  Examples:
  - **sync unbuffered** E.g. Blocking read-call, `fread()` or SPI-read, zero-copying in shared memory.
  - **sync buffered** E.g. Blocking read-call with caching applied. Makes sense if read pattern exhibits spatial locality.
  - **async unbuffered** E.g. zero-copying in shared memory. Be careful of data hazards, race conditions, etc!
  - **async buffered** E.g. `aio_read()` or DMA.

Unfortunately, I/O may not be atomic and may, therefore, require locking. Locking may not be needed if multiple independent channels are available.

- **The system has non-atomic write and one shared channel** E.g. UART. Locking required.

  ```
  lock(); emit(a); emit(b); emit(c); release();
  ```

- **The system has non-atomic write but many channels** E.g. Multi-UART. Lock-free if the bottom-layer maps each Zephyr thread+ISR to its own channel, thus alleviating races as each thread is sequentially consistent with itself.

  ```
  emit(a,thread_id); emit(b,thread_id); emit(c,thread_id);
  ```

- **The system has atomic write but one shared channel** E.g. `native_posix` or board with DMA. May or may not need locking.

  ```
  emit(a ## b ## c); /* Concat to buffer */

  lock(); emit(a); emit(b); emit(c); release(); /* No extra mem */
  ```

- **The system has atomic write and many channels** E.g. native_posix or board with multi-channel DMA. Lock-free.

  ```
  emit(a ## b ## c, thread_id);
  ```

API

Common

group `tracing_apis`

Tracing APIs.

Functions

void `sys_trace_isr_enter`(void)
Called when entering an ISR.

void `sys_trace_isr_exit`(void)
Called when exiting an ISR.

void `sys_trace_isr_exit_to_scheduler`(void)
Called when exiting an ISR and switching to scheduler.

void `sys_trace_idle`(void)
Called when the cpu enters the idle state.

Threads

group `thread_tracing_apis`
Thread Tracing APIs.

Defines

`sys_port_trace_k_thread_foreach_enter`()
Called when entering a k_thread_foreach call.

`sys_port_trace_k_thread_foreach_exit`()
Called when exiting a k_thread_foreach call.

`sys_port_trace_k_thread_foreach_unlocked_enter`()
Called when entering a k_thread_foreach_unlocked.

`sys_port_trace_k_thread_foreach_unlocked_exit`()
Called when exiting a k_thread_foreach_unlocked.

`sys_port_trace_k_thread_create`(new_thread)
Trace creating a Thread.

Parameters

- `new_thread` – Thread object

`sys_port_trace_k_thread_user_mode_enter`()
Trace Thread entering user mode.

`sys_port_trace_k_thread_join_enter`(thread, timeout)
Called when entering a k_thread_join.

Parameters

- `thread` – Thread object
- `timeout` – Timeout period

`sys_port_trace_k_thread_join_blocking`(thread, timeout)
Called when k_thread_join blocks.

Parameters

- `thread` – Thread object
- `timeout` – Timeout period

sys_port_trace_k_thread_join_exit(thread, timeout, ret)
Called when exiting k_thread_join.

Parameters

- thread – Thread object
- timeout – Timeout period
- ret – Return value

sys_port_trace_k_thread_sleep_enter(timeout)
Called when entering k_thread_sleep.

Parameters

- timeout – Timeout period

sys_port_trace_k_thread_sleep_exit(timeout, ret)
Called when exiting k_thread_sleep.

Parameters

- timeout – Timeout period
- ret – Return value

sys_port_trace_k_thread_msleep_enter(ms)
Called when entering k_thread_msleep.

Parameters

- ms – Duration in milliseconds

sys_port_trace_k_thread_msleep_exit(ms, ret)
Called when exiting k_thread_msleep.

Parameters

- ms – Duration in milliseconds
- ret – Return value

sys_port_trace_k_thread_usleep_enter(us)
Called when entering k_thread_usleep.

Parameters

- us – Duration in microseconds

sys_port_trace_k_thread_usleep_exit(us, ret)
Called when exiting k_thread_usleep.

Parameters

- us – Duration in microseconds
- ret – Return value

sys_port_trace_k_thread_busy_wait_enter(usec_to_wait)
Called when entering k_thread_busy_wait.

Parameters

- usec_to_wait – Duration in microseconds

sys_port_trace_k_thread_busy_wait_exit(usec_to_wait)
Called when exiting k_thread_busy_wait.

Parameters

- `usec_to_wait` – Duration in microseconds

`sys_port_trace_k_thread_yield()`
Called when a thread yields.

`sys_port_trace_k_thread_wakeup(thread)`
Called when a thread wakes up.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_start(thread)`
Called when a thread is started.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_abort(thread)`
Called when a thread is being aborted.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_abort_enter(thread)`
Called when a thread enters the k_thread_abort routine.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_abort_exit(thread)`
Called when a thread exits the k_thread_abort routine.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_priority_set(thread)`
Called when setting priority of a thread.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_suspend_enter(thread)`
Called when a thread enters the k_thread_suspend function.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_suspend_exit(thread)`
Called when a thread exits the k_thread_suspend function.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_resume_enter(thread)`
Called when a thread enters the resume from suspension function.

Parameters

- `thread` – Thread object

sys_port_trace_k_thread_resume_exit(thread)
Called when a thread exits the resumed from suspension function.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_lock()
Called when the thread scheduler is locked.

sys_port_trace_k_thread_sched_unlock()
Called when the thread sceduler is unlocked.

sys_port_trace_k_thread_name_set(thread, ret)
Called when a thread name is set.

Parameters

- thread – Thread object
- ret – Return value

sys_port_trace_k_thread_switched_out()
Called before a thread has been selected to run.

sys_port_trace_k_thread_switched_in()
Called after a thread has been selected to run.

sys_port_trace_k_thread_ready(thread)
Called when a thread is ready to run.

Parameters

- thread – Thread object

sys_port_trace_k_thread_pend(thread)
Called when a thread is pending.

Parameters

- thread – Thread object

sys_port_trace_k_thread_info(thread)
Provide information about specific thread.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_wakeup(thread)
Trace implicit thread wakup invocation by the scheduler.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_abort(thread)
Trace implicit thread abort invocation by the scheduler.

Parameters

- thread – Thread object

sys_port_trace_k_thread_sched_priority_set(thread, prio)
Trace implicit thread set priority invocation by the scheduler.

Parameters

- thread – Thread object

- `prio` – Thread priority

`sys_port_trace_k_thread_sched_ready(thread)`
Trace implicit thread ready invocation by the scheduler.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_sched_pend(thread)`
Trace implicit thread pend invocation by the scheduler.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_sched_resume(thread)`
Trace implicit thread resume invocation by the scheduler.

Parameters

- `thread` – Thread object

`sys_port_trace_k_thread_sched_suspend(thread)`
Trace implicit thread suspend invocation by the scheduler.

Parameters

- `thread` – Thread object

Work Queues

group `work_tracing_apis`
Work Tracing APIs.

Defines

`sys_port_trace_k_work_init(work)`
Trace initialisation of a Work structure.

Parameters

- `work` – Work structure

`sys_port_trace_k_work_submit_to_queue_enter(queue, work)`
Trace submit work to work queue call entry.

Parameters

- `queue` – Work queue structure
- `work` – Work structure

`sys_port_trace_k_work_submit_to_queue_exit(queue, work, ret)`
Trace submit work to work queue call exit.

Parameters

- `queue` – Work queue structure
- `work` – Work structure
- `ret` – Return value

sys_port_trace_k_work_submit_enter(work)
Trace submit work to system work queue call entry.

Parameters

- work – Work structure

sys_port_trace_k_work_submit_exit(work, ret)
Trace submit work to system work queue call exit.

Parameters

- work – Work structure
- ret – Return value

sys_port_trace_k_work_flush_enter(work)
Trace flush work call entry.

Parameters

- work – Work structure

sys_port_trace_k_work_flush_blocking(work, timeout)
Trace flush work call blocking.

Parameters

- work – Work structure
- timeout – Timeout period

sys_port_trace_k_work_flush_exit(work, ret)
Trace flush work call exit.

Parameters

- work – Work structure
- ret – Return value

sys_port_trace_k_work_cancel_enter(work)
Trace cancel work call entry.

Parameters

- work – Work structure

sys_port_trace_k_work_cancel_exit(work, ret)
Trace cancel work call exit.

Parameters

- work – Work structure
- ret – Return value

sys_port_trace_k_work_cancel_sync_enter(work, sync)
Trace cancel sync work call entry.

Parameters

- work – Work structure
- sync – Sync object

sys_port_trace_k_work_cancel_sync_blocking(work, sync)
Trace cancel sync work call blocking.

Parameters

- `work` – Work structure
- `sync` – Sync object

`sys_port_trace_k_work_cancel_sync_exit(work, sync, ret)`
Trace cancel sync work call exit.

Parameters

- `work` – Work structure
- `sync` – Sync object
- `ret` – Return value

Poll

group `poll_tracing_apis`
Poll Tracing APIs.

Defines

`sys_port_trace_k_poll_api_event_init(event)`
Trace initialisation of a Poll Event.

Parameters

- `event` – Poll Event

`sys_port_trace_k_poll_api_poll_enter(events)`
Trace Polling call start.

Parameters

- `events` – Poll Events

`sys_port_trace_k_poll_api_poll_exit(events, ret)`
Trace Polling call outcome.

Parameters

- `events` – Poll Events
- `ret` – Return value

`sys_port_trace_k_poll_api_signal_init(signal)`
Trace initialisation of a Poll Signal.

Parameters

- `signal` – Poll Signal

`sys_port_trace_k_poll_api_signal_reset(signal)`
Trace resetting of Poll Signal.

Parameters

- `signal` – Poll Signal

`sys_port_trace_k_poll_api_signal_check(signal)`
Trace checking of Poll Signal.

Parameters

- `signal` – Poll Signal

sys_port_trace_k_poll_api_signal_raise(signal, ret)

Trace raising of Poll Signal.

Parameters

- signal – Poll Signal
- ret – Return value

Semaphore

group sem_tracing_apis

Semaphore Tracing APIs.

Defines

sys_port_trace_k_sem_init(sem, ret)

Trace initialisation of a Semaphore.

Parameters

- sem – Semaphore object
- ret – Return value

sys_port_trace_k_sem_give_enter(sem)

Trace giving a Semaphore entry.

Parameters

- sem – Semaphore object

sys_port_trace_k_sem_give_exit(sem)

Trace giving a Semaphore exit.

Parameters

- sem – Semaphore object

sys_port_trace_k_sem_take_enter(sem, timeout)

Trace taking a Semaphore attempt start.

Parameters

- sem – Semaphore object
- timeout – Timeout period

sys_port_trace_k_sem_take_blocking(sem, timeout)

Trace taking a Semaphore attempt blocking.

Parameters

- sem – Semaphore object
- timeout – Timeout period

sys_port_trace_k_sem_take_exit(sem, timeout, ret)

Trace taking a Semaphore attempt outcome.

Parameters

- sem – Semaphore object
- timeout – Timeout period
- ret – Return value

`sys_port_trace_k_sem_reset(sem)`
Trace resetting a Semaphore.

Parameters

- `sem` – Semaphore object

Mutex

group `mutex_tracing_apis`
Mutex Tracing APIs.

Defines

`sys_port_trace_k_mutex_init(mutex, ret)`
Trace initialization of Mutex.

Parameters

- `mutex` – Mutex object
- `ret` – Return value

`sys_port_trace_k_mutex_lock_enter(mutex, timeout)`
Trace Mutex lock attempt start.

Parameters

- `mutex` – Mutex object
- `timeout` – Timeout period

`sys_port_trace_k_mutex_lock_blocking(mutex, timeout)`
Trace Mutex lock attempt blocking.

Parameters

- `mutex` – Mutex object
- `timeout` – Timeout period

`sys_port_trace_k_mutex_lock_exit(mutex, timeout, ret)`
Trace Mutex lock attempt outcome.

Parameters

- `mutex` – Mutex object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mutex_unlock_enter(mutex)`
Trace Mutex unlock entry.

Parameters

- `mutex` – Mutex object

`sys_port_trace_k_mutex_unlock_exit(mutex, ret)`
Trace Mutex unlock exit.

Condition Variables

group `condvar_tracing_apis`
Conditional Variable Tracing APIs.

Defines

sys_port_trace_k_condvar_init(condvar, ret)

Trace initialization of Conditional Variable.

Parameters

- condvar – Conditional Variable object
- ret – Return value

sys_port_trace_k_condvar_signal_enter(condvar)

Trace Conditional Variable signaling start.

Parameters

- condvar – Conditional Variable object

sys_port_trace_k_condvar_signal_blocking(condvar, timeout)

Trace Conditional Variable signaling blocking.

Parameters

- condvar – Conditional Variable object
- timeout – Timeout period

sys_port_trace_k_condvar_signal_exit(condvar, ret)

Trace Conditional Variable signaling outcome.

Parameters

- condvar – Conditional Variable object
- ret – Return value

sys_port_trace_k_condvar_broadcast_enter(condvar)

Trace Conditional Variable broadcast enter.

Parameters

- condvar – Conditional Variable object

sys_port_trace_k_condvar_broadcast_exit(condvar, ret)

Trace Conditional Variable broadcast exit.

Parameters

- condvar – Conditional Variable object
- ret – Return value

sys_port_trace_k_condvar_wait_enter(condvar)

Trace Conditional Variable wait enter.

Parameters

- condvar – Conditional Variable object

sys_port_trace_k_condvar_wait_exit(condvar, ret)

Trace Conditional Variable wait exit.

Parameters

- condvar – Conditional Variable object
- ret – Return value

Queues

group `queue_tracing_apis`

Queue Tracing APIs.

Defines

`sys_port_trace_k_queue_init(queue)`

Trace initialization of Queue.

Parameters

- `queue` – Queue object

`sys_port_trace_k_queue_cancel_wait(queue)`

Trace Queue cancel wait.

Parameters

- `queue` – Queue object

`sys_port_trace_k_queue_queue_insert_enter(queue, alloc)`

Trace Queue insert attempt entry.

Parameters

- `queue` – Queue object
- `alloc` – Allocation flag

`sys_port_trace_k_queue_queue_insert_blocking(queue, alloc, timeout)`

Trace Queue insert attempt blocking.

Parameters

- `queue` – Queue object
- `alloc` – Allocation flag
- `timeout` – Timeout period

`sys_port_trace_k_queue_queue_insert_exit(queue, alloc, ret)`

Trace Queue insert attempt outcome.

Parameters

- `queue` – Queue object
- `alloc` – Allocation flag
- `ret` – Return value

`sys_port_trace_k_queue_append_enter(queue)`

Trace Queue append enter.

Parameters

- `queue` – Queue object

`sys_port_trace_k_queue_append_exit(queue)`

Trace Queue append exit.

Parameters

- `queue` – Queue object

sys_port_trace_k_queue_alloc_append_enter(queue)

Trace Queue alloc append enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_alloc_append_exit(queue, ret)

Trace Queue alloc append exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_prepend_enter(queue)

Trace Queue prepend enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_prepend_exit(queue)

Trace Queue prepend exit.

Parameters

- queue – Queue object

sys_port_trace_k_queue_alloc_prepend_enter(queue)

Trace Queue alloc prepend enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_alloc_prepend_exit(queue, ret)

Trace Queue alloc prepend exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_insert_enter(queue)

Trace Queue insert attempt entry.

Parameters

- queue – Queue object

sys_port_trace_k_queue_insert_blocking(queue, timeout)

Trace Queue insert attempt blocking.

Parameters

- queue – Queue object
- timeout – Timeout period

sys_port_trace_k_queue_insert_exit(queue)

Trace Queue insert attempt exit.

Parameters

- queue – Queue object

sys_port_trace_k_queue_append_list_enter(queue)
Trace Queue append list enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_append_list_exit(queue, ret)
Trace Queue append list exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_merge_slist_enter(queue)
Trace Queue merge slist enter.

Parameters

- queue – Queue object

sys_port_trace_k_queue_merge_slist_exit(queue, ret)
Trace Queue merge slist exit.

Parameters

- queue – Queue object
- ret – Return value

sys_port_trace_k_queue_get_enter(queue, timeout)
Trace Queue get attempt enter.

Parameters

- queue – Queue object
- timeout – Timeout period

sys_port_trace_k_queue_get_blocking(queue, timeout)
Trace Queue get attempt blockings.

Parameters

- queue – Queue object
- timeout – Timeout period

sys_port_trace_k_queue_get_exit(queue, timeout, ret)
Trace Queue get attempt outcome.

Parameters

- queue – Queue object
- timeout – Timeout period
- ret – Return value

sys_port_trace_k_queue_remove_enter(queue)
Trace Queue remove enter.

Parameters

- queue – Queue object

`sys_port_trace_k_queue_remove_exit`(queue, ret)

Trace Queue remove exit.

Parameters

- `queue` – Queue object
- `ret` – Return value

`sys_port_trace_k_queue_unique_append_enter`(queue)

Trace Queue unique append enter.

Parameters

- `queue` – Queue object

`sys_port_trace_k_queue_unique_append_exit`(queue, ret)

Trace Queue unique append exit.

Parameters

- `queue` – Queue object
- `ret` – Return value

`sys_port_trace_k_queue_peek_head`(queue, ret)

Trace Queue peek head.

Parameters

- `queue` – Queue object
- `ret` – Return value

`sys_port_trace_k_queue_peek_tail`(queue, ret)

Trace Queue peek tail.

Parameters

- `queue` – Queue object
- `ret` – Return value

FIFO

group `fifo_tracing_apis`

FIFO Tracing APIs.

Defines

`sys_port_trace_k_fifo_init_enter`(fifo)

Trace initialization of FIFO Queue entry.

Parameters

- `fifo` – FIFO object

`sys_port_trace_k_fifo_init_exit`(fifo)

Trace initialization of FIFO Queue exit.

Parameters

- `fifo` – FIFO object

sys_port_trace_k_fifo_cancel_wait_enter(fifo)

Trace FIFO Queue cancel wait entry.

Parameters

- fifo – FIFO object

sys_port_trace_k_fifo_cancel_wait_exit(fifo)

Trace FIFO Queue cancel wait exit.

Parameters

- fifo – FIFO object

sys_port_trace_k_fifo_put_enter(fifo, data)

Trace FIFO Queue put entry.

Parameters

- fifo – FIFO object
- data – Data item

sys_port_trace_k_fifo_put_exit(fifo, data)

Trace FIFO Queue put exit.

Parameters

- fifo – FIFO object
- data – Data item

sys_port_trace_k_fifo_alloc_put_enter(fifo, data)

Trace FIFO Queue alloc put entry.

Parameters

- fifo – FIFO object
- data – Data item

sys_port_trace_k_fifo_alloc_put_exit(fifo, data, ret)

Trace FIFO Queue alloc put exit.

Parameters

- fifo – FIFO object
- data – Data item
- ret – Return value

sys_port_trace_k_fifo_alloc_put_list_enter(fifo, head, tail)

Trace FIFO Queue put list entry.

Parameters

- fifo – FIFO object
- head – First ll-node
- tail – Last ll-node

sys_port_trace_k_fifo_alloc_put_list_exit(fifo, head, tail)

Trace FIFO Queue put list exit.

Parameters

- fifo – FIFO object
- head – First ll-node

- `tail` – Last ll-node

`sys_port_trace_k_fifo_alloc_put_slist_enter`(fifo, list)

Trace FIFO Queue put slist entry.

Parameters

- `fifo` – FIFO object
- `list` – Syslist object

`sys_port_trace_k_fifo_alloc_put_slist_exit`(fifo, list)

Trace FIFO Queue put slist exit.

Parameters

- `fifo` – FIFO object
- `list` – Syslist object

`sys_port_trace_k_fifo_get_enter`(fifo, timeout)

Trace FIFO Queue get entry.

Parameters

- `fifo` – FIFO object
- `timeout` – Timeout period

`sys_port_trace_k_fifo_get_exit`(fifo, timeout, ret)

Trace FIFO Queue get exit.

Parameters

- `fifo` – FIFO object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_fifo_peek_head_entry`(fifo)

Trace FIFO Queue peek head entry.

Parameters

- `fifo` – FIFO object

`sys_port_trace_k_fifo_peek_head_exit`(fifo, ret)

Trace FIFO Queue peek head exit.

Parameters

- `fifo` – FIFO object
- `ret` – Return value

`sys_port_trace_k_fifo_peek_tail_entry`(fifo)

Trace FIFO Queue peek tail entry.

Parameters

- `fifo` – FIFO object

`sys_port_trace_k_fifo_peek_tail_exit`(fifo, ret)

Trace FIFO Queue peek tail exit.

Parameters

- `fifo` – FIFO object
- `ret` – Return value

LIFO

group `lifo_tracing_apis`

LIFO Tracing APIs.

Defines

`sys_port_trace_k_lifo_init_enter`(lifo)

Trace initialization of LIFO Queue entry.

Parameters

- `lifo` – LIFO object

`sys_port_trace_k_lifo_init_exit`(lifo)

Trace initialization of LIFO Queue exit.

Parameters

- `lifo` – LIFO object

`sys_port_trace_k_lifo_put_enter`(lifo, data)

Trace LIFO Queue put entry.

Parameters

- `lifo` – LIFO object
- `data` – Data item

`sys_port_trace_k_lifo_put_exit`(lifo, data)

Trace LIFO Queue put exit.

Parameters

- `lifo` – LIFO object
- `data` – Data item

`sys_port_trace_k_lifo_alloc_put_enter`(lifo, data)

Trace LIFO Queue alloc put entry.

Parameters

- `lifo` – LIFO object
- `data` – Data item

`sys_port_trace_k_lifo_alloc_put_exit`(lifo, data, ret)

Trace LIFO Queue alloc put exit.

Parameters

- `lifo` – LIFO object
- `data` – Data item
- `ret` – Return value

`sys_port_trace_k_lifo_get_enter`(lifo, timeout)

Trace LIFO Queue get entry.

Parameters

- `lifo` – LIFO object
- `timeout` – Timeout period

sys_port_trace_k_lifo_get_exit(lifo, timeout, ret)

Trace LIFO Queue get exit.

Parameters

- lifo – LIFO object
- timeout – Timeout period
- ret – Return value

Stacks

group stack_tracing_apis

Stack Tracing APIs.

Defines

sys_port_trace_k_stack_init(stack)

Trace initialization of Stack.

Parameters

- stack – Stack object

sys_port_trace_k_stack_alloc_init_enter(stack)

Trace Stack alloc init attempt entry.

Parameters

- stack – Stack object

sys_port_trace_k_stack_alloc_init_exit(stack, ret)

Trace Stack alloc init outcome.

Parameters

- stack – Stack object
- ret – Return value

sys_port_trace_k_stack_cleanup_enter(stack)

Trace Stack cleanup attempt entry.

Parameters

- stack – Stack object

sys_port_trace_k_stack_cleanup_exit(stack, ret)

Trace Stack cleanup outcome.

Parameters

- stack – Stack object
- ret – Return value

sys_port_trace_k_stack_push_enter(stack)

Trace Stack push attempt entry.

Parameters

- stack – Stack object

`sys_port_trace_k_stack_push_exit`(stack, ret)

Trace Stack push attempt outcome.

Parameters

- `stack` – Stack object
- `ret` – Return value

`sys_port_trace_k_stack_pop_enter`(stack, timeout)

Trace Stack pop attempt entry.

Parameters

- `stack` – Stack object
- `timeout` – Timeout period

`sys_port_trace_k_stack_pop_blocking`(stack, timeout)

Trace Stack pop attempt blocking.

Parameters

- `stack` – Stack object
- `timeout` – Timeout period

`sys_port_trace_k_stack_pop_exit`(stack, timeout, ret)

Trace Stack pop attempt outcome.

Parameters

- `stack` – Stack object
- `timeout` – Timeout period
- `ret` – Return value

Message Queues

group `msgq_tracing_apis`

Message Queue Tracing APIs.

Defines

`sys_port_trace_k_msgq_init`(msgq)

Trace initialization of Message Queue.

Parameters

- `msgq` – Message Queue object

`sys_port_trace_k_msgq_alloc_init_enter`(msgq)

Trace Message Queue alloc init attempt entry.

Parameters

- `msgq` – Message Queue object

`sys_port_trace_k_msgq_alloc_init_exit`(msgq, ret)

Trace Message Queue alloc init attempt outcome.

Parameters

- `msgq` – Message Queue object
- `ret` – Return value

`sys_port_trace_k_msgq_cleanup_enter(msgq)`
Trace Message Queue cleanup attempt entry.

Parameters

- `msgq` – Message Queue object

`sys_port_trace_k_msgq_cleanup_exit(msgq, ret)`
Trace Message Queue cleanup attempt outcome.

Parameters

- `msgq` – Message Queue object
- `ret` – Return value

`sys_port_trace_k_msgq_put_enter(msgq, timeout)`
Trace Message Queue put attempt entry.

Parameters

- `msgq` – Message Queue object
- `timeout` – Timeout period

`sys_port_trace_k_msgq_put_blocking(msgq, timeout)`
Trace Message Queue put attempt blocking.

Parameters

- `msgq` – Message Queue object
- `timeout` – Timeout period

`sys_port_trace_k_msgq_put_exit(msgq, timeout, ret)`
Trace Message Queue put attempt outcome.

Parameters

- `msgq` – Message Queue object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_msgq_get_enter(msgq, timeout)`
Trace Message Queue get attempt entry.

Parameters

- `msgq` – Message Queue object
- `timeout` – Timeout period

`sys_port_trace_k_msgq_get_blocking(msgq, timeout)`
Trace Message Queue get attempt blockings.

Parameters

- `msgq` – Message Queue object
- `timeout` – Timeout period

`sys_port_trace_k_msgq_get_exit(msgq, timeout, ret)`
Trace Message Queue get attempt outcome.

Parameters

- `msgq` – Message Queue object
- `timeout` – Timeout period

- `ret` – Return value

`sys_port_trace_k_msgq_peek(msgq, ret)`
Trace Message Queue peek.

Parameters

- `msgq` – Message Queue object
- `ret` – Return value

`sys_port_trace_k_msgq_purge(msgq)`
Trace Message Queue purge.

Parameters

- `msgq` – Message Queue object

Mailbox

group `mbox_tracing_apis`
Mailbox Tracing APIs.

Defines

`sys_port_trace_k_mbox_init(mbox)`
Trace initialization of Mailbox.

Parameters

- `mbox` – Mailbox object

`sys_port_trace_k_mbox_message_put_enter(mbox, timeout)`
Trace Mailbox message put attempt entry.

Parameters

- `mbox` – Mailbox object
- `timeout` – Timeout period

`sys_port_trace_k_mbox_message_put_blocking(mbox, timeout)`
Trace Mailbox message put attempt blocking.

Parameters

- `mbox` – Mailbox object
- `timeout` – Timeout period

`sys_port_trace_k_mbox_message_put_exit(mbox, timeout, ret)`
Trace Mailbox message put attempt outcome.

Parameters

- `mbox` – Mailbox object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mbox_put_enter(mbox, timeout)`
Trace Mailbox put attempt entry.

Parameters

- `mbox` – Mailbox object

- `timeout` – Timeout period

`sys_port_trace_k_mbox_put_exit`(mbox, timeout, ret)

Trace Mailbox put attempt blocking.

Parameters

- `mbox` – Mailbox object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mbox_async_put_enter`(mbox, sem)

Trace Mailbox async put entry.

Parameters

- `mbox` – Mailbox object
- `sem` – Semaphore object

`sys_port_trace_k_mbox_async_put_exit`(mbox, sem)

Trace Mailbox async put exit.

Parameters

- `mbox` – Mailbox object
- `sem` – Semaphore object

`sys_port_trace_k_mbox_get_enter`(mbox, timeout)

Trace Mailbox get attempt entry.

Parameters

- `mbox` – Mailbox entry
- `timeout` – Timeout period

`sys_port_trace_k_mbox_get_blocking`(mbox, timeout)

Trace Mailbox get attempt blocking.

Parameters

- `mbox` – Mailbox entry
- `timeout` – Timeout period

`sys_port_trace_k_mbox_get_exit`(mbox, timeout, ret)

Trace Mailbox get attempt outcome.

Parameters

- `mbox` – Mailbox entry
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mbox_data_get`(rx_msg)

Trace Mailbox data get.

rx_msg Receive Message object

Pipes

group `pipe_tracing_apis`

Pipe Tracing APIs.

Defines

`sys_port_trace_k_pipe_init(pipe)`
Trace initialization of Pipe.

Parameters

- `pipe` – Pipe object

`sys_port_trace_k_pipe_cleanup_enter(pipe)`
Trace Pipe cleanup entry.

Parameters

- `pipe` – Pipe object

`sys_port_trace_k_pipe_cleanup_exit(pipe, ret)`
Trace Pipe cleanup exit.

Parameters

- `pipe` – Pipe object
- `ret` – Return value

`sys_port_trace_k_pipe_alloc_init_enter(pipe)`
Trace Pipe alloc init entry.

Parameters

- `pipe` – Pipe object

`sys_port_trace_k_pipe_alloc_init_exit(pipe, ret)`
Trace Pipe alloc init exit.

Parameters

- `pipe` – Pipe object
- `ret` – Return value

`sys_port_trace_k_pipe_put_enter(pipe, timeout)`
Trace Pipe put attempt entry.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_put_blocking(pipe, timeout)`
Trace Pipe put attempt blocking.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_put_exit(pipe, timeout, ret)`
Trace Pipe put attempt outcome.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_pipe_get_enter`(pipe, timeout)
Trace Pipe get attempt entry.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_get_blocking`(pipe, timeout)
Trace Pipe get attempt blocking.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period

`sys_port_trace_k_pipe_get_exit`(pipe, timeout, ret)
Trace Pipe get attempt outcome.

Parameters

- `pipe` – Pipe object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_pipe_block_put_enter`(pipe, sem)
Trace Pipe block put enter.

Parameters

- `pipe` – Pipe object
- `sem` – Semaphore object

`sys_port_trace_k_pipe_block_put_exit`(pipe, sem)
Trace Pipe block put exit.

Parameters

- `pipe` – Pipe object
- `sem` – Semaphore object

Heaps

group `heap_tracing_apis`
Heap Tracing APIs.

Defines

`sys_port_trace_k_heap_init`(h)
Trace initialization of Heap.

Parameters

- `h` – Heap object

`sys_port_trace_k_heap_aligned_alloc_enter`(h, timeout)
Trace Heap aligned alloc attempt entry.

Parameters

- `h` – Heap object

- `timeout` – Timeout period

`sys_port_trace_k_heap_aligned_alloc_blocking(h, timeout)`
Trace Heap align alloc attempt blocking.

Parameters

- `h` – Heap object
- `timeout` – Timeout period

`sys_port_trace_k_heap_aligned_alloc_exit(h, timeout, ret)`
Trace Heap align alloc attempt outcome.

Parameters

- `h` – Heap object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_heap_alloc_enter(h, timeout)`
Trace Heap alloc enter.

Parameters

- `h` – Heap object
- `timeout` – Timeout period

`sys_port_trace_k_heap_alloc_exit(h, timeout, ret)`
Trace Heap alloc exit.

Parameters

- `h` – Heap object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_heap_free(h)`
Trace Heap free.

Parameters

- `h` – Heap object

`sys_port_trace_k_heap_sys_k_aligned_alloc_enter(heap)`
Trace System Heap aligned alloc enter.

Parameters

- `heap` – Heap object

`sys_port_trace_k_heap_sys_k_aligned_alloc_exit(heap, ret)`
Trace System Heap aligned alloc exit.

Parameters

- `heap` – Heap object
- `ret` – Return value

`sys_port_trace_k_heap_sys_k_malloc_enter(heap)`
Trace System Heap aligned alloc enter.

Parameters

- `heap` – Heap object

`sys_port_trace_k_heap_sys_k_malloc_exit`(heap, ret)
Trace System Heap aligned alloc exit.

Parameters

- `heap` – Heap object
- `ret` – Return value

`sys_port_trace_k_heap_sys_k_free_enter`(heap)
Trace System Heap free entry.

Parameters

- `heap` – Heap object

`sys_port_trace_k_heap_sys_k_free_exit`(heap)
Trace System Heap free exit.

Parameters

- `heap` – Heap object

`sys_port_trace_k_heap_sys_k_calloc_enter`(heap)
Trace System heap calloc enter.

Parameters

- `heap` –

`sys_port_trace_k_heap_sys_k_calloc_exit`(heap, ret)
Trace System heap calloc exit.

Parameters

- `heap` – Heap object
- `ret` – Return value

Memory Slabs

group `mslab_tracing_apis`
Memory Slab Tracing APIs.

Defines

`sys_port_trace_k_mem_slab_init`(slab, rc)
Trace initialization of Memory Slab.

Parameters

- `slab` – Memory Slab object
- `rc` – Return value

`sys_port_trace_k_mem_slab_alloc_enter`(slab, timeout)
Trace Memory Slab alloc attempt entry.

Parameters

- `slab` – Memory Slab object
- `timeout` – Timeout period

`sys_port_trace_k_mem_slab_alloc_blocking`(slab, timeout)
Trace Memory Slab alloc attempt blocking.

Parameters

- `slab` – Memory Slab object
- `timeout` – Timeout period

`sys_port_trace_k_mem_slab_alloc_exit`(slab, timeout, ret)
Trace Memory Slab alloc attempt outcome.

Parameters

- `slab` – Memory Slab object
- `timeout` – Timeout period
- `ret` – Return value

`sys_port_trace_k_mem_slab_free_enter`(slab)
Trace Memory Slab free entry.

Parameters

- `slab` – Memory Slab object

`sys_port_trace_k_mem_slab_free_exit`(slab)
Trace Memory Slab free exit.

Parameters

- `slab` – Memory Slab object

Timers

group `timer_tracing_apis`
Timer Tracing APIs.

Defines

`sys_port_trace_k_timer_init`(timer)
Trace initialization of Timer.

Parameters

- `timer` – Timer object

`sys_port_trace_k_timer_start`(timer)
Trace Timer start.

Parameters

- `timer` – Timer object

`sys_port_trace_k_timer_stop`(timer)
Trace Timer stop.

Parameters

- `timer` – Timer object

`sys_port_trace_k_timer_status_sync_enter`(timer)

Trace Timer status sync entry.

Parameters

- `timer` – Timer object

`sys_port_trace_k_timer_status_sync_blocking`(timer, timeout)

Trace Timer Status sync blocking.

Parameters

- `timer` – Timer object
- `timeout` – Timeout period

`sys_port_trace_k_timer_status_sync_exit`(timer, result)

Trace Time Status sync outcome.

Parameters

- `timer` – Timer object
- `result` – Return value

8.10 Device Management

8.10.1 MCUmgr

Overview

The management subsystem allows remote management of Zephyr-enabled devices. The following management operations are available:

- Image management
- File System management
- Log management (currently disabled)
- OS management
- Shell management

over the following transports:

- BLE (Bluetooth Low Energy)
- Serial (UART)
- UDP over IP

The management subsystem is based on the Simple Management Protocol (SMP) provided by MCUmgr, an open source project that provides a management subsystem that is portable across multiple real-time operating systems.

The management subsystem is split in two different locations in the Zephyr tree:

- zephyrproject-rtos/mcumgr repo contains a clean import of the MCUmgr project
- subsys/mgmt/ contains the Zephyr-specific bindings to MCUmgr

Additionally there is a sample that provides management functionality over BLE and serial.

Command-line Tool

MCUmgr provides a command-line tool, `mcumgr`, for managing remote devices. The tool is written in the Go programming language.

To install the tool:

```
go get github.com/apache/mynewt-mcumgr-cli/mcumgr
```

Configuring the transport

There are two command-line options that are responsible for setting and configuring the transport layer to use when communicating with managed device:

- `--conntype` is used to choose the transport used, and
- `--connstring` is used to pass a comma separated list of options in the `key=value` format, where each valid `key` depends on the particular `conntype`.

Valid transports for `--conntype` are `serial`, `ble` and `udp`. Each transport expects a different set of key/value options:

serial

`--connstring` accepts the following `key` values:

`dev`	the device name for the OS `mcumgr` is running on (eg, `/dev/ttyUSB0`, `/dev/tty.usbserial`, `COM1`, etc).
`baud`	the communication speed; must match the baudrate of the server.
`mtu`	aka Maximum Transmission Unit, the maximum protocol packet size.

ble

`--connstring` accepts the following `key` values:

`ctlr_name`	an OS specific string for the controller name.
`own_addr_type`	can be one of `public`, `random`, `rpa_pub`, `rpa_rnd`, where `random` is the default.
`peer_name`	the name the peer BLE device advertises, this should match the configuration specified with `CONFIG_BT_DEVICE_NAME`.
`peer_id`	the peer BLE device address or UUID. Only required when `peer_name` was not given. The format depends on the OS where `mcumgr` is run, it is a 6 bytes hexadecimal string separated by colons on Linux, or a 128-bit UUID on macOS.
`conn_timeout`	a float number representing the connection timeout in seconds.

udp

`--connstring` takes the form `[addr]:port` where:

`addr`	can be a DNS name (if it can be resolved to the device IP), IPv4 addr (app must be built with `CONFIG_MCUMGR_SMP_UDP_IPV4`), or IPv6 addr (app must be built with `CONFIG_MCUMGR_SMP_UDP_IPV6`)
`port`	any valid UDP port.

Saving the connection config

The transport configuration can be managed with the `conn` sub-command and later used with `--conn` (or `-c`) parameter to skip typing both `--conntype` and `--connstring`. For example a new config for a serial device that would require typing `mcumgr --conntype serial --connstring dev=/dev/ttyACM0,baud=115200,mtu=512` can be saved with:

```
mcumgr conn add acm0 type="serial" connstring="dev=/dev/ttyACM0,baud=115200,mtu=512"
```

Accessing this port can now be done with:

```
mcumgr -c acm0
```

General options

Some options work for every `mcumgr` command and might be helpful to debug and fix issues with the communication, among them the following deserve special mention:

`-l <log-level>`	Configures the log level, which can be one of `critical`, `error`, `warn`, `info` or `debug`, from less to most verbose. When there are communication issues, `-lDEBUG` might be useful to dump the packets for later inspection.
`-t <timeout>`	Changes the timeout waiting for a response from the default of 10s to a given value. Some commands might take a long time of processing, eg, the erase before an image upload, and might need incrementing the timeout to a larger value.
`-r <tries>`	Changes the number of retries on timeout from the default of 1 to a given value.

List of Commands

Not all commands defined by `mcumgr` (and SMP protocol) are currently supported on Zephyr. The ones that are supported are described in the following table:

Tip: Running `mcumgr` with no parameters, or `-h` will display the list of commands.

Command	Description
`echo`	Send data to a device and display the echoed back data. This command is part of the `OS` group, which must be enabled by setting `CONFIG_MCUMGR_CMD_OS_MGMT`. The `echo` command itself can be enabled by setting `CONFIG_OS_MGMT_ECHO`.
`fs`	Access files on a device. More info in *Filesystem Management*.
`image`	Manage images on a device. More info in *Image Management*.
`reset`	Perform a soft reset of a device. This command is part of the `OS` group, which must be enabled by setting `CONFIG_MCUMGR_CMD_OS_MGMT`. The `reset` command itself is always enabled and the time taken for a reset to happen can be set with `CONFIG_OS_MGMT_RESET_MS` (in ms).
`shell`	Execute a command in the remote shell. This option is disabled by default and can be enabled with `CONFIG_MCUMGR_CMD_SHELL_MGMT = y`. To know more about the shell in Zephyr check *Shell*.
`stat`	Read statistics from a device. More info in *Statistics Management*.
`taskstat`	Read task statistics from a device. This command is part of the `OS` group, which must be enabled by setting `CONFIG_MCUMGR_CMD_OS_MGMT`. The `taskstat` command itself can be enabled by setting `CONFIG_OS_MGMT_TASKSTAT`. `CONFIG_THREAD_MONITOR` also needs to be enabled otherwise a `-8` (`MGMT_ERR_ENOTSUP`) will be returned.

Tip: `taskstat` has a few options that might require tweaking. The `CONFIG_THREAD_NAME` must be set to display the task names, otherwise the priority is displayed. Since the `taskstat` packets are large, they might need increasing the `CONFIG_MCUMGR_BUF_SIZE` option.

Warning: To display the correct stack size in the `taskstat` command, the `CONFIG_THREAD_STACK_INFO` option must be set. To display the correct stack usage in the `taskstat` command, both `CONFIG_THREAD_STACK_INFO` and `CONFIG_INIT_STACKS` options must be set.

J-Link Virtual MSD Interaction Note

On boards where a J-Link OB is present which has both CDC and MSC (virtual Mass Storage Device, also known as drag-and-drop) support, the MSD functionality can prevent mcumgr commands over the CDC UART port from working due to how USB endpoints are configured in the J-Link firmware (for example on the Nordic `nrf52840dk`) because of limiting the maximum packet size (most likely to occur when using image management commands for updating firmware). This issue can be resolved by disabling MSD functionality on the J-Link device, follow the instructions on nordic_segger_msd to disable MSD support.

Image Management

The image management provided by `mcumgr` is based on the image format defined by MCUboot. For more details on the internals see MCUboot design and *Signing Binaries*.

To list available images in a device:

```
mcumgr <connection-options> image list
```

This should result in an output similar to this:

```
$ mcumgr -c acm0 image list
Images:
  image=0 slot=0
    version: 1.0.0
    bootable: true
    flags: active confirmed
    hash: 86dca73a3439112b310b5e033d811ec2df728d2264265f2046fced5a9ed00cc7
Split status: N/A (0)
```

Where `image` is the number of the image pair in a multi-image system, and slot is the number of the slot where the image is stored, `0` for primary and `1` for secondary. This image being `active` and `confirmed` means it will run again on next reset. Also relevant is the `hash`, which is used by other commands to refer to this specific image when performing operations.

An image can be manually erased using:

```
mcumgr <connection-options> image erase
```

The behavior of `erase` is defined by the server (`mcumgr` in the device). The current implementation is limited to erasing the image in the secondary partition.

To upload a new image:

```
mcumgr <connection-options> image upload [-n] [-e] [-u] <signed-bin>
```

- `-n`: This option allows uploading a new image to a specific set of images in a multi-image system, and is currently only supported by MCUboot when the CONFIG_MCUBOOT_SERIAL option is enabled.
- `-e`: This option avoids performing a full erase of the partition before starting a new upload.

Tip: The `-e` option should always be passed in because the `upload` command already checks if an erase is required, respecting the `CONFIG_IMG_ERASE_PROGRESSIVELY` setting.

Tip: If the `upload` command times out while waiting for a response from the device, `-t` might be used to increase the wait time to something larger than the default of 10s. See *general_options*.

Warning: `mcumgr` does not understand .hex files, when uploading a new image always use the .bin file.

- `-u`: upgrade only to newer image version.

After an image upload is finished, a new `image list` would now have an output like this:

```
$ mcumgr -c acm0 image upload -e build/zephyr/zephyr.signed.bin
  35.69 KiB / 92.92 KiB [==========>---------------]  38.41% 2.97 KiB/s 00m19
```

Now listing the images again:

```
$ mcumgr -c acm0 image list
Images:
 image=0 slot=0
  version: 1.0.0
  bootable: true
  flags: active confirmed
  hash: 86dca73a3439112b310b5e033d811ec2df728d2264265f2046fced5a9ed00cc7
 image=0 slot=1
  version: 1.1.0
  bootable: true
  flags:
  hash: e8cf0dcef3ec8addee07e8c4d5dc89e64ba3fae46a2c5267fc4efbea4ca0e9f4
Split status: N/A (0)
```

To test a new upgrade image the `test` command is used:

```
mcumgr <connection-options> image test <hash>
```

This command should mark a `test` upgrade, which means that after the next reboot the bootloader will execute the upgrade and jump into the new image. If no other image operations are executed on the newly running image, it will `revert` back to the image that was previously running on the device on the subsequent reset. When a `test` is requested, `flags` will be updated with `pending` to inform that a new image will be run after a reset:

```
$ mcumgr -c acm0 image test␣
↪e8cf0dcef3ec8addee07e8c4d5dc89e64ba3fae46a2c5267fc4efbea4ca0e9f4
Images:
 image=0 slot=0
  version: 1.0.0
  bootable: true
  flags: active confirmed
  hash: 86dca73a3439112b310b5e033d811ec2df728d2264265f2046fced5a9ed00cc7
 image=0 slot=1
  version: 1.1.0
  bootable: true
  flags: pending
```

(continues on next page)

(continued from previous page)

```
  hash: e8cf0dcef3ec8addee07e8c4d5dc89e64ba3fae46a2c5267fc4efbea4ca0e9f4
Split status: N/A (0)
```

After a reset the output with change to:

```
$ mcumgr -c acm0 image list
Images:
 image=0 slot=0
  version: 1.1.0
  bootable: true
  flags: active
  hash: e8cf0dcef3ec8addee07e8c4d5dc89e64ba3fae46a2c5267fc4efbea4ca0e9f4
 image=0 slot=1
  version: 1.0.0
  bootable: true
  flags: confirmed
  hash: 86dca73a3439112b310b5e033d811ec2df728d2264265f2046fced5a9ed00cc7
Split status: N/A (0)
```

Tip: It's important to mention that an upgrade only ever happens if the image is valid. The first thing MCUboot does when an upgrade is requested is to validate the image, using the SHA-256 and/or the signature (depending on the configuration). So before uploading an image, one way to be sure it is valid is to run `imgtool verify -k <your-signature-key> <your-image>`, where `-k <your-signature-key` can be skipped if no signature validation was enabled.

The `confirmed` flag in the secondary slot tells that after the next reset a revert upgrade will be perfomed to switch back to the original layout.

The command used to confirm that an image is OK and no revert should happen (no `hash` required) is:

```
mcumgr <connection-options> image confirm [hash]
```

The `confirm` command can also be run passing in a `hash` so that instead of doing a `test/revert` procedure, the image in the secondary partition is directly upgraded to.

Tip: The whole `test/revert` cycle does not need to be done using only the `mcumgr` command-line tool. A better alternative is to perform a `test` and allow the new running image to self-confirm after any checks by calling `boot_write_img_confirmed()`.

Tip: The maximum size of a chunk communicated between the client and server is set with `CONFIG_IMG_MGMT_UL_CHUNK_SIZE`. The default is 512 but can be decreased for systems with low amount of RAM downto 128. When this value is changed, the `mtu` of the port must be smaller than or equal to this value.

Tip: Building with `CONFIG_IMG_MGMT_VERBOSE_ERR` enables better error messages when failures happen (but increases the application size).

Statistics Management

Statistics are used for troubleshooting, maintenance, and usage monitoring; it consists basically of user-defined counters which are tightly connected to `mcumgr` and can be used to track any information for

easy retrieval. The available sub-commands are:

```
mcumgr <connection-options> stat list
mcumgr <connection-options> stat <section-name>
```

Statistics are organized in sections (also called groups), and each section can be individually queried. Defining new statistics sections is done by using macros available under `<stats/stats.h>`. Each section consists of multiple variables (or counters), all with the same size (16, 32 or 64 bits).

To create a new section `my_stats`:

```
STATS_SECT_START(my_stats)
  STATS_SECT_ENTRY(my_stat_counter1)
  STATS_SECT_ENTRY(my_stat_counter2)
  STATS_SECT_ENTRY(my_stat_counter3)
STATS_SECT_END;

STATS_SECT_DECL(my_stats) my_stats;
```

Each entry can be declared with `STATS_SECT_ENTRY` (or the equivalent `STATS_SECT_ENTRY32`), `STATS_SECT_ENTRY16` or `STATS_SECT_ENTRY64`. All statistics in a section must be declared with the same size.

The statistics counters can either have names or not, depending on the setting of the `CONFIG_STATS_NAMES` option. Using names requires an extra declaration step:

```
STATS_NAME_START(my_stats)
  STATS_NAME(my_stats, my_stat_counter1)
  STATS_NAME(my_stats, my_stat_counter2)
  STATS_NAME(my_stats, my_stat_counter3)
STATS_NAME_END(my_stats);
```

Tip: Disabling `CONFIG_STATS_NAMES` will free resources. When this option is disabled the `STATS_NAME*` macros output nothing, so adding them in the code does not increase the binary size.

Tip: `CONFIG_STAT_MGMT_MAX_NAME_LEN` sets the maximum length of a section name that can can be accepted as parameter for showing the section data, and might require tweaking for long section names.

The final steps to use a statistics section is to initialize and register it:

```
rc = STATS_INIT_AND_REG(my_stats, STATS_SIZE_32, "my_stats");
assert (rc == 0);
```

In the running code a statistics counter can be incremented by 1 using `STATS_INC`, by N using `STATS_INCN` or reset with `STATS_CLEAR`.

Let's suppose we want to increment those counters by `1`, `2` and `3` every second. To get a list of stats:

```
$ mcumgr --conn acm0 stat list
stat groups:
  my_stats
```

To get the current value of the counters in `my_stats`:

```
$ mcumgr --conn acm0 stat my_stats
stat group: my_stats
      13 my_stat_counter1
```

(continues on next page)

(continued from previous page)

```
      26 my_stat_counter2
      39 my_stat_counter3

$ mcumgr --conn acm0 stat my_stats
stat group: my_stats
      16 my_stat_counter1
      32 my_stat_counter2
      48 my_stat_counter3
```

When `CONFIG_STATS_NAMES` is disabled the output will look like this:

```
$ mcumgr --conn acm0 stat my_stats
stat group: my_stats
       8 s0
      16 s1
      24 s2
```

Filesystem Management

The filesystem module is disabled by default due to security concerns: because of a lack of access control every file in the FS will be accessible, including secrets, etc. To enable it `CONFIG_MCUMGR_CMD_FS_MGMT` must be set (`y`). Once enabled the following sub-commands can be used:

```
mcumgr <connection-options> fs download <remote-file> <local-file>
mcumgr <connection-options> fs upload <local-file> <remote-file>
```

Using the `fs` command, requires `CONFIG_FILE_SYSTEM` to be enabled, and that some particular filesystem is enabled and properly mounted by the running application, eg for littefs this would mean enabling `CONFIG_FILE_SYSTEM_LITTLEFS`, defining a storage partition *Flash map* and mounting the filesystem in the startup (*`fs_mount()`*).

Uploading a new file to a littlefs storage, mounted under `/lfs`, can be done with:

```
$ mcumgr -c acm0 fs upload foo.txt /lfs/foo.txt
25
Done
```

Where `25` is the size of the file.

For downloading a file, let's first use the `fs` command (`CONFIG_FILE_SYSTEM_SHELL` must be enabled) in a remote shell to create a new file:

```
uart:~$ fs write /lfs/bar.txt 41 42 43 44 31 32 33 34 0a
uart:~$ fs read /lfs/bar.txt
File size: 9
00000000  41 42 43 44 31 32 33 34 0A                        ABCD1234.
```

Now it can be downloaded using:

```
$ mcumgr -c acm0 fs download /lfs/bar.txt bar.txt
0
9
Done
$ cat bar.txt
ABCD1234
```

Where `0` is the return code, and `9` is the size of the file.

Warning: The commands might exhaust the system workqueue, if its size is not large enough, so increasing `CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE` might be required for correct behavior.

The size of the stack allocated buffer used to store the blocks, while transffering a file can be adjusted with `CONFIG_FS_MGMT_DL_CHUNK_SIZE`; this allows saving RAM resources.

Tip: `CONFIG_FS_MGMT_PATH_SIZE` sets the maximum PATH accepted for a file name. It might require tweaking for longer file names.

Bootloader integration

The *Device Firmware Upgrade* subsystem integrates the management subsystem with the bootloader, providing the ability to send and upgrade a Zephyr image to a device.

Currently only the MCUboot bootloader is supported. See *MCUboot* for more information.

8.10.2 Device Firmware Upgrade

Overview

The Device Firmware Upgrade subsystem provides the necessary frameworks to upgrade the image of a Zephyr-based application at run time. It currently consists of two different modules:

- subsys/dfu/boot/: Interface code to bootloaders
- subsys/dfu/img_util/: Image management code

The DFU subsystem deals with image management, but not with the transport or management protocols themselves required to send the image to the target device. For information on these protocols and frameworks please refer to the *Device Management* section.

Bootloaders

MCUboot Zephyr is directly compatible with the open source, cross-RTOS MCUboot boot loader. It interfaces with MCUboot and is aware of the image format required by it, so that Device Firmware Upgrade is available when MCUboot is the boot loader used with Zephyr. The source code itself is hosted in the MCUboot GitHub Project page.

In order to use MCUboot with Zephyr you need to take the following into account:

1. You will need to define the flash partitions required by MCUboot; see *Flash map* for details.
2. You will have to specify your flash parition as the chosen code partition

```
/ {
   chosen {
      zephyr,code-partition = &slot0_partition;
   };
};
```

3. Your application's `.conf` file needs to enable the `CONFIG_BOOTLOADER_MCUBOOT` Kconfig option in order for Zephyr to be built in an MCUboot-compatible manner
4. You need to build and flash MCUboot itself on your device
5. You might need to take precautions to avoid mass erasing the flash and also to flash the Zephyr application image at the correct offset (right after the bootloader)

More detailed information regarding the use of MCUboot with Zephyr can be found in the MCUboot with Zephyr documentation page on the MCUboot website.

8.11 Devicetree Guide

This is a high-level guide to devicetree as it is used for Zephyr development. See *Devicetree* for reference material.

8.11.1 Introduction to devicetree

Tip: This is a conceptual overview of devicetree and how Zephyr uses it. For step-by-step guides and examples, see *Devicetree HOWTOs*.

A *devicetree* is a hierarchical data structure that describes hardware. The Devicetree specification defines its source and binary representations. Zephyr uses devicetree to describe the hardware available on its boards, as well as that hardware's initial configuration.

There are two types of devicetree input files: *devicetree sources* and *devicetree bindings*. The sources contain the devicetree itself. The bindings describe its contents, including data types. The *build system* uses devicetree sources and bindings to produce a generated C header. The generated header's contents are abstracted by the `devicetree.h` API, which you can use to get information from your devicetree.

Here is a simplified view of the process:

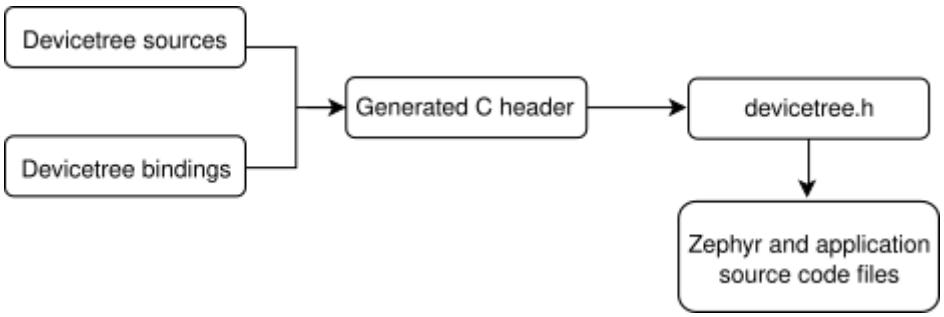


Fig. 5: Devicetree build flow

All Zephyr and application source code files can include and use `devicetree.h`. This includes *device drivers*, *applications*, *tests*, the kernel, etc.

The API itself is based on C macros. The macro names all start with `DT_`. In general, if you see a macro that starts with `DT_` in a Zephyr source file, it's probably a `devicetree.h` macro. The generated C header contains macros that start with `DT_` as well; you might see those in compiler error messages. You always can tell a generated- from a non-generated macro: generated macros have some lowercased letters, while the `devicetree.h` macro names have all capital letters.

Some information defined in devicetree is available via `CONFIG_` macros generated from *Kconfig*. This is often done for backwards compatibility, since Zephyr has used Kconfig for longer than devicetree, and is still in the process of converting some information from Kconfig to devicetree. It is also done to allow Kconfig overrides of default values taken from devicetree. Devicetree information is referenced from Kconfig via *Kconfig functions*. See *Devicetree versus Kconfig* for some additional comparisons with Kconfig.

Syntax and structure

As the name indicates, a devicetree is a tree. The human-readable text format for this tree is called DTS (for devicetree source), and is defined in the Devicetree specification.

Here is an example DTS file:

```
/dts-v1/;

/ {
        a-node {
                subnode_label: a-sub-node {
                        foo = <3>;
                };
        };
};
```

The `/dts-v1/;` line means the file's contents are in version 1 of the DTS syntax, which has replaced a now-obsolete "version 0".

The tree has three *nodes*:

1. A root node: `/`
2. A node named `a-node`, which is a child of the root node
3. A node named `a-sub-node`, which is a child of `a-node`

Nodes can be given *labels*, which are unique shorthands that can be used to refer to the labeled node elsewhere in the devicetree. Above, `a-sub-node` has label `subnode_label`. A node can have zero, one, or multiple node labels.

Devicetree nodes have *paths* identifying their locations in the tree. Like Unix file system paths, devicetree paths are strings separated by slashes (`/`), and the root node's path is a single slash: `/`. Otherwise, each node's path is formed by concatenating the node's ancestors' names with the node's own name, separated by slashes. For example, the full path to `a-sub-node` is `/a-node/a-sub-node`.

Devicetree nodes can also have *properties*. Properties are name/value pairs. Property values can be any sequence of bytes. In some cases, the values are an array of what are called *cells*. A cell is just a 32-bit unsigned integer.

Node `a-sub-node` has a property named `foo`, whose value is a cell with value 3. The size and type of `foo`'s value are implied by the enclosing angle brackets (< and >) in the DTS. See *Writing property values* below for more example property values.

In practice, devicetree nodes usually correspond to some hardware, and the node hierarchy reflects the hardware's physical layout. For example, let's consider a board with three I2C peripherals connected to an I2C bus controller on an SoC, like this:

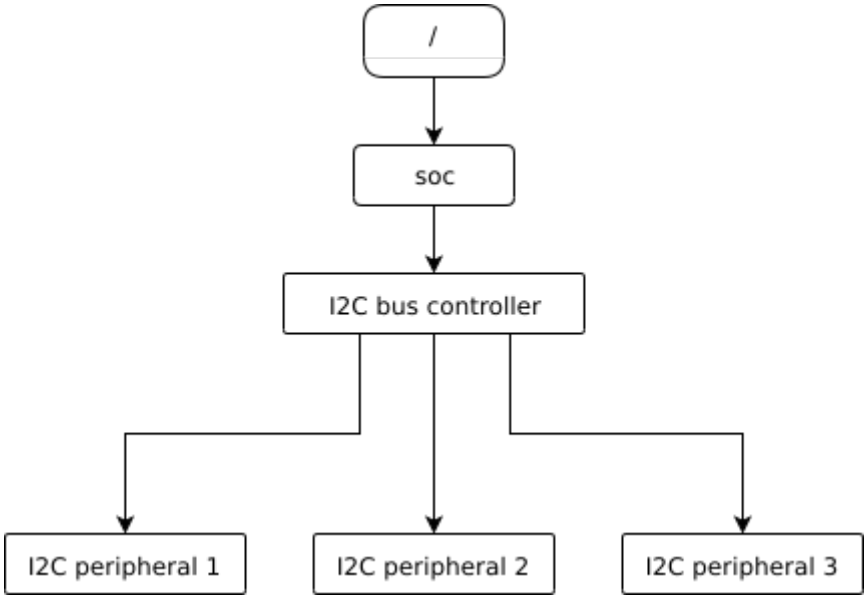
Nodes corresponding to the I2C bus controller and each I2C peripheral would be present in the devicetree. Reflecting the hardware layout, the I2C peripheral nodes would be children of the bus controller node. Similar conventions exist for representing other types of hardware.

The DTS would look something like this:

```
/dts-v1/;

/ {
        soc {
                i2c-bus-controller {
                        i2c-peripheral-1 {
                        };
                        i2c-peripheral-2 {
                        };
```

(continues on next page)



(continued from previous page)

```
                        i2c-peripheral-3 {
                        };
                };
        };
};
```

Properties are used in practice to describe or configure the hardware the node represents. For example, an I2C peripheral's node has a property whose value is the peripheral's address on the bus.

Here's a tree representing the same example, but with real-world node names and properties you might see when working with I2C devices.

This is the corresponding DTS:

```
/dts-v1/;

/ {
        soc {
                i2c@40003000 {
                        compatible = "nordic,nrf-twim";
                        label = "I2C_0";
                        reg = <0x40003000 0x1000>;

                        apds9960@39 {
                                compatible = "avago,apds9960";
                                label = "APDS9960";
                                reg = <0x39>;
                        };
                        ti_hdc@43 {
                                compatible = "ti,hdc", "ti,hdc1010";
                                label = "HDC1010";
                                reg = <0x43>;
                        };
                        mma8652fc@1d {
                                compatible = "nxp,fxos8700", "nxp,mma8652fc";
                                label = "MMA8652FC";
                                reg = <0x1d>;
```

(continues on next page)

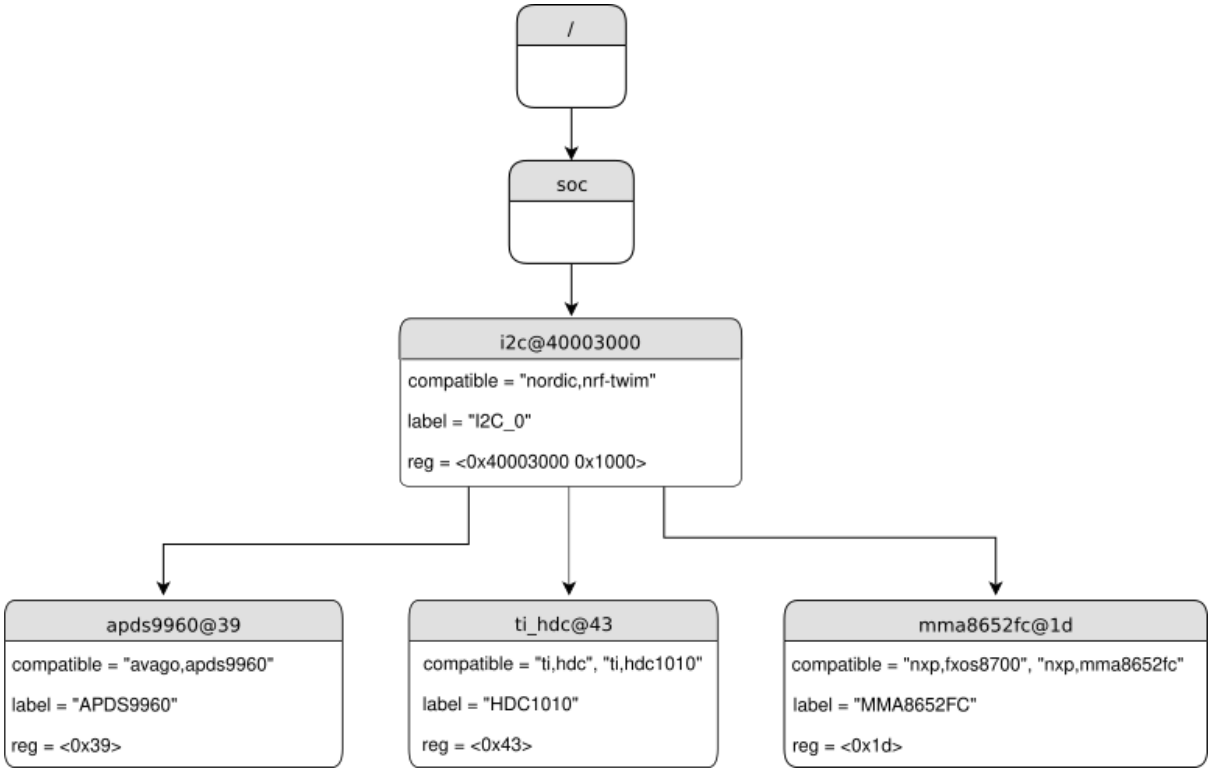


Fig. 6: I2C devicetree example with real-world names and properties. Node names are at the top of each node with a gray background. Properties are shown as "name=value" lines.

(continued from previous page)

```
                        };
                };
        };
};
```

In addition to showing more realistic names and properties, the above example introduces a new devicetree concept: unit addresses. Unit addresses are the parts of node names after an "at" sign (`@`), like `40003000` in `i2c@40003000`, or `39` in `apds9960@39`. Unit addresses are optional: the `soc` node does not have one.

Some more details about unit addresses and important properties follow.

Unit address examples

In devicetree, unit addresses give a node's address in the address space of its parent node. Here are some example unit addresses for different types of hardware.

Memory-mapped peripherals The peripheral's register map base address. For example, the node named `i2c@40003000` represents an I2C controller whose register map base address is 0x40003000.

I2C peripherals The peripheral's address on the I2C bus. For example, the child node `apds9960@39` of the I2C controller in the previous section has I2C address 0x39.

SPI peripherals An index representing the peripheral's chip select line number. (If there is no chip select line, 0 is used.)

Memory The physical start address. For example, a node named `memory@2000000` represents RAM starting at physical address 0x2000000.

Memory-mapped flash Like RAM, the physical start address. For example, a node named `flash@8000000` represents a flash device whose physical start address is 0x8000000.

Fixed flash partitions This applies when the devicetree is used to store a flash partition table. The unit address is the partition's start offset within the flash memory. For example, take this flash device and its partitions:

```
flash@8000000 {
    /* ... */
    partitions {
            partition@0 { /* ... */ };
            partition@20000 {  /* ... */ };
            /* ... */
    };
};
```

The node named `partition@0` has offset 0 from the start of its flash device, so its base address is 0x8000000. Similarly, the base address of the node named `partition@20000` is 0x8020000.

Important properties

Some important properties are:

compatible The name of the hardware device the node represents.

The recommended format is `"vendor,device"`, like `"avago,apds9960"`, or a sequence of these, like `"ti,hdc", "ti,hdc1010"`. The `vendor` part is an abbreviated name of the vendor. The file dts/bindings/vendor-prefixes.txt contains a list of commonly accepted `vendor` names. The `device` part is usually taken from the datasheet.

It is also sometimes a value like `gpio-keys`, `mmio-sram`, or `fixed-clock` when the hardware's behavior is generic.

The build system uses the compatible property to find the right *bindings* for the node. Device drivers use `devicetree.h` to find nodes with relevant compatibles, in order to determine the available hardware to manage.

The `compatible` property can have multiple values. Additional values are useful when the device is a specific instance of a more general family, to allow the system to match from most- to least-specific device drivers.

Within Zephyr's bindings syntax, this property has type `string-array`.

label The device's name according to Zephyr's *Device Driver Model*. The value can be passed to *`device_get_binding()`* to retrieve the corresponding driver-level *struct device**. This pointer can then be passed to the correct driver API by application code to interact with the device. For example, calling `device_get_binding("I2C_0")` would return a pointer to a device structure which could be passed to *I2C API* functions like *`i2c_transfer()`*. The generated C header will also contain a macro which expands to this string.

reg Information used to address the device. The value is specific to the device (i.e. is different depending on the compatible property).

The `reg` property is a sequence of `(address, length)` pairs. Each pair is called a "register block". Here are some common patterns:

- Devices accessed via memory-mapped I/O registers (like `i2c@40003000`): `address` is usually the base address of the I/O register space, and `length` is the number of bytes occupied by the registers.
- I2C devices (like `apds9960@39` and its siblings): `address` is a slave address on the I2C bus. There is no `length` value.
- SPI devices: `address` is a chip select line number; there is no `length`.

You may notice some similarities between the `reg` property and common unit addresses described above. This is not a coincidence. The `reg` property can be seen as a more detailed view of the addressable resources within a device than its unit address.

status A string which describes whether the node is enabled.

The devicetree specification allows this property to have values `"okay"`, `"disabled"`, `"reserved"`, `"fail"`, and `"fail-sss"`. Only the values `"okay"` and `"disabled"` are currently relevant to Zephyr; use of other values currently results in undefined behavior.

A node is considered enabled if its status property is either `"okay"` or not defined (i.e. does not exist in the devicetree source). Nodes with status `"disabled"` are explicitly disabled. (For backwards compatibility, the value `"ok"` is treated the same as `"okay"`, but this usage is deprecated.) Devicetree nodes which correspond to physical devices must be enabled for the corresponding `struct device` in the Zephyr driver model to be allocated and initialized.

interrupts Information about interrupts generated by the device, encoded as an array of one or more *interrupt specifiers*. Each interrupt specifier has some number of cells. See section 2.4, *Interrupts and Interrupt Mapping*, in the Devicetree Specification release v0.3 for more details.

Zephyr's devicetree bindings language lets you give a name to each cell in an interrupt specifier.

Writing property values

This section describes how to write property values in DTS format. The property types in the table below are described in detail in *Devicetree bindings*.

Some specifics are skipped in the interest of keeping things simple; if you're curious about details, see the devicetree specification.

Property type	How to write	Example
string	Double quoted	`a-string = "hello, world!";`
int	between angle brackets (< and >)	`an-int = <1>;`
boolean	for `true`, with no value (for `false`, use `/delete-property/`)	`my-true-boolean;`
array	between angle brackets (< and >), separated by spaces	`foo = <0xdeadbeef 1234 0>;`
uint8-array	in hexadecimal *without* leading `0x`, between square brackets (`[` and `]`).	`a-byte-array = [00 01 ab];`
string-array	separated by commas	`a-string-array = "string one", "string two", "string three";`
phandle	between angle brackets (< and >)	`a-phandle = <&mynode>;`
phandles	between angle brackets (< and >), separated by spaces	`some-phandles = <&mynode0 &mynode1 &mynode2>;`
phandle-array	between angle brackets (< and >), separated by spaces	`a-phandle-array = <&mynode0 1 2 &mynode1 3 4>;`

Additional notes on the above:

- Boolean properties are true if present. They should not have a value. A boolean property is only false if it is completely missing in the DTS.
- The `foo` property value above has three *cells* with values 0xdeadbeef, 1234, and 0, in that order. Note that hexadecimal and decimal numbers are allowed and can be intermixed. Since Zephyr transforms DTS to C sources, it is not necessary to specify the endianness of an individual cell here.
- 64-bit integers are written as two 32-bit cells in big-endian order. The value 0xaaaa0000bbbb1111 would be written `<0xaaaa0000 0xbbbb1111>`.
- The `a-byte-array` property value is the three bytes 0x00, 0x01, and 0xab, in that order.

- Parentheses, arithmetic operators, and bitwise operators are allowed. The `bar` property contains a single cell with value 64:

  ```
  bar = <(2 * (1 << 5))>;
  ```

  Note that the entire expression must be parenthesized.

- Property values refer to other nodes in the devicetree by their *phandles*. You can write a phandle using `&foo`, where `foo` is a *node label*. Here is an example devicetree fragment:

  ```
  foo: device@0 { };
  device@1 {
          sibling = <&foo 1 2>;
  };
  ```

  The `sibling` property of node `device@1` contains three cells, in this order:

  1. The `device@0` node's phandle, which is written here as `&foo` since the `device@0` node has a node label `foo`
  2. The value 1
  3. The value 2

  In the devicetree, a phandle value is a cell – which again is just a 32-bit unsigned int. However, the Zephyr devicetree API generally exposes these values as *node identifiers*. Node identifiers are covered in more detail in *Devicetree access from C/C++*.

- Array and similar type property values can be split into several <> blocks, like this:

  ```
  foo = <1 2>, <3 4>;                          // Okay for 'type: array'
  foo = <&label1 &label2>, <&label3 &label4>; // Okay for 'type: phandles'
  foo = <&label1 1 2>, <&label2 3 4>;          // Okay for 'type: phandle-array'
  ```

  This is recommended for readability when possible if the value can be logically grouped into blocks of sub-values.

Aliases and chosen nodes

There are two additional ways beyond *node labels* to refer to a particular node without specifying its entire path: by alias, or by chosen node.

Here is an example devicetree which uses both:

```
/dts-v1/;

/ {
     chosen {
             zephyr,console = &uart0;
     };

     aliases {
             my-uart = &uart0;
     };

     soc {
             uart0: serial@12340000 {
                     ...
             };
     };
};
```

The `/aliases` and `/chosen` nodes do not refer to an actual hardware device. Their purpose is to specify other nodes in the devicetree.

Above, `my-uart` is an alias for the node with path `/soc/serial@12340000`. Using its node label `uart0`, the same node is set as the value of the chosen `zephyr,console` node.

Zephyr sample applications sometimes use aliases to allow overriding the particular hardware device used by the application in a generic way. For example, blinky-sample uses this to abstract the LED to blink via the `led0` alias.

The `/chosen` node's properties are used to configure system- or subsystem-wide values. See *Chosen nodes* for more information.

Input and output files

This section describes the input and output files shown in the figure at the *top of this introduction* in more detail.

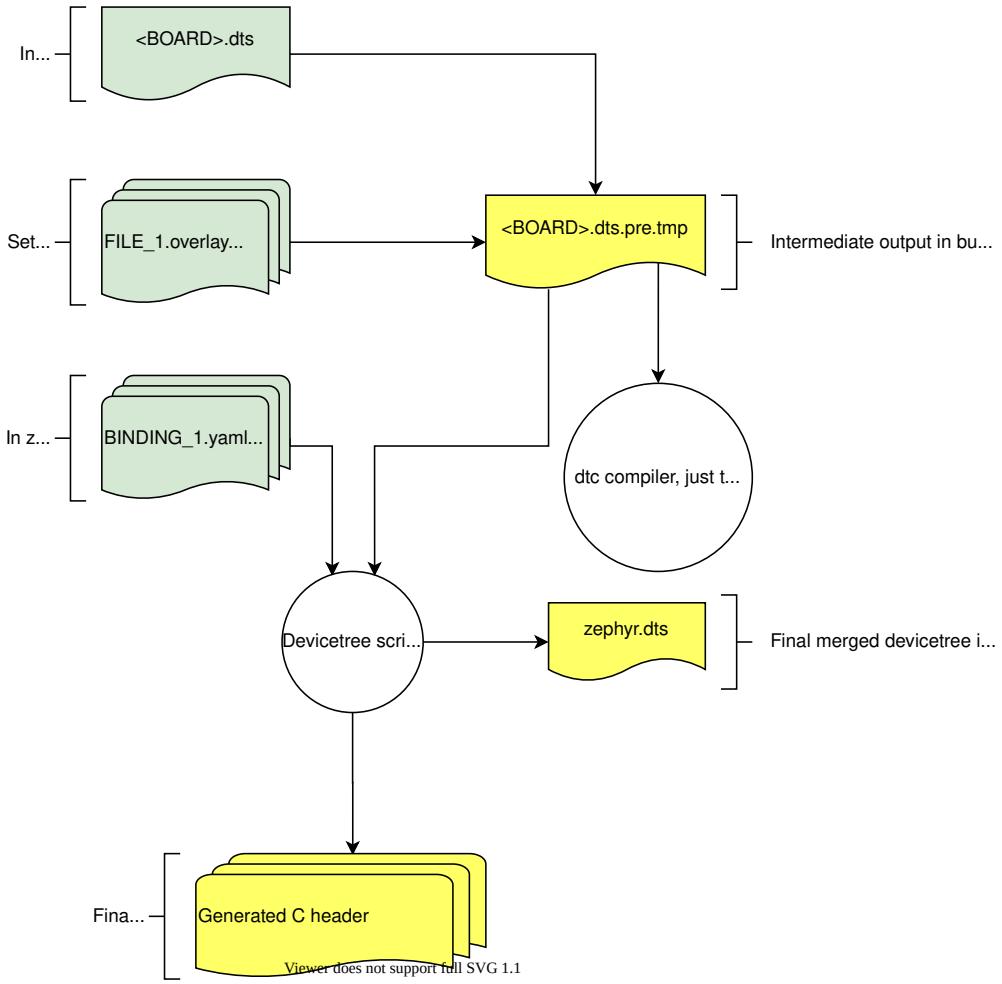


Fig. 7: Devicetree input (green) and output (yellow) files

Input files There are four types of devicetree input files:

- sources (`.dts`)
- includes (`.dtsi`)
- overlays (`.overlay`)

- bindings (`.yaml`)

The devicetree files inside the `zephyr` directory look like this:

```
boards/<ARCH>/<BOARD>/<BOARD>.dts
dts/common/skeleton.dtsi
dts/<ARCH>/.../<SOC>.dtsi
dts/bindings/.../binding.yaml
```

Generally speaking, every supported board has a `BOARD.dts` file describing its hardware. For example, the `reel_board` has boards/arm/reel_board/reel_board.dts.

`BOARD.dts` includes one or more `.dtsi` files. These `.dtsi` files describe the CPU or system-on-chip Zephyr runs on, perhaps by including other `.dtsi` files. They can also describe other common hardware features shared by multiple boards. In addition to these includes, `BOARD.dts` also describes the board's specific hardware.

The `dts/common` directory contains `skeleton.dtsi`, a minimal include file for defining a complete devicetree. Architecture-specific subdirectories (`dts/<ARCH>`) contain `.dtsi` files for CPUs or SoCs which extend `skeleton.dtsi`.

The C preprocessor is run on all devicetree files to expand macro references, and includes are generally done with `#include <filename>` directives, even though DTS has a `/include/ "<filename>"` syntax.

`BOARD.dts` can be extended or modified using *overlays*. Overlays are also DTS files; the `.overlay` extension is just a convention which makes their purpose clear. Overlays adapt the base devicetree for different purposes:

- Zephyr applications can use overlays to enable a peripheral that is disabled by default, select a sensor on the board for an application specific purpose, etc. Along with *Configuration System (Kconfig)*, this makes it possible to reconfigure the kernel and device drivers without modifying source code.
- Overlays are also used when defining *Shields*.

The build system automatically picks up `.overlay` files stored in certain locations. It is also possible to explicitly list the overlays to include, via the `DTC_OVERLAY_FILE` CMake variable. See *Set devicetree overlays* for details.

The build system combines `BOARD.dts` and any `.overlay` files by concatenating them, with the overlays put last. This relies on DTS syntax which allows merging overlapping definitions of nodes in the devicetree. See *Example: FRDM-K64F and Hexiwear K64* for an example of how this works (in the context of `.dtsi` files, but the principle is the same for overlays). Putting the contents of the `.overlay` files last allows them to override `BOARD.dts`.

Devicetree bindings (which are YAML files) are essentially glue. They describe the contents of devicetree sources, includes, and overlays in a way that allows the build system to generate C macros usable by device drivers and applications. The `dts/bindings` directory contains bindings.

Zephyr currently uses `dts_fixup.h` files to rename macros in `devicetree_unfixed.h` to names that are currently in use by C code. The build system looks for fixup files in the `zephyr/boards/` and `zephyr/soc/` directories by default. Fixup files exist for historical reasons. New code should generally avoid them.

Scripts and tools The following libraries and scripts, located in scripts/dts/, create output files from input files. Their sources have extensive documentation.

dtlib.py A low-level DTS parsing library.

edtlib.py A library layered on top of dtlib that uses bindings to interpret properties and give a higher-level view of the devicetree. Uses dtlib to do the DTS parsing.

gen_defines.py A script that uses edtlib to generate C preprocessor macros from the devicetree and bindings.

In addition to these, the standard `dtc` (devicetree compiler) tool is run on the final devicetree if it is installed on your system. This is just to catch errors or warnings. The output is unused. Boards may need to pass `dtc` additional flags, e.g. for warning suppression. Board directories can contain a file named `pre_dt_board.cmake` which configures these extra flags, like this:

```
list(APPEND EXTRA_DTC_FLAGS "-Wno-simple_bus_reg")
```

Output files These are created in your application's build directory.

Warning: Don't include the header files directly. *Devicetree access from C/C++* explains what to do instead.

`<build>/zephyr/include/generated/devicetree_unfixed.h` The generated macros and additional comments describing the devicetree. Included by `devicetree.h`.

`<build>/zephyr/include/generated/devicetree_fixups.h` The concatenated contents of any `dts_fixup.h` files. Included by `devicetree.h`.

`<build>/zephyr/zephyr.dts` The final merged devicetree. This file is output by `gen_defines.py` as a debugging aid, and is unused otherwise.

`<build>/zephyr/<BOARD>.dts.pre.tmp` The preprocessed and concatenated DTS sources and overlays. This is an intermediate output file, which is used to create `zephyr.dts` and `devicetree_unfixed.h`.

8.11.2 Design goals

Zephyr's use of devicetree has evolved significantly over time, and further changes are expected. The following are the general design goals, along with specific examples about how they impact Zephyr's source code, and areas where more work remains to be done.

Single source for all hardware information

Zephyr shall obtain its hardware descriptions exclusively from devicetree.

Examples

- New device drivers shall use devicetree APIs to determine which *devices to create* if possible.
- In-tree sample applications shall use *aliases* to determine which of multiple possible generic devices of a given type will be used in the current build. For example, the blinky-sample uses this to determine the LED to blink.
- Boot-time pin muxing and pin control can be accomplished via devicetree.

Example remaining work

- Zephyr's *Test Runner (Twister)* currently use `board.yaml` files to determine the hardware supported by a board. This should be obtained from devicetree instead.
- Various device drivers currently use Kconfig to determine which instances of a particular compatible are enabled. This can and should be done with devicetree overlays instead.
- Board-level documentation still contains tables of hardware support which are generated and maintained by hand. This can and should be obtained from the board level devicetree instead.
- Runtime determination of `struct device` relationships should be done using information obtained from devicetree, e.g. for device power management.

Source compatibility with other operating systems

Zephyr's devicetree tooling is based on a generic layer which is interoperable with other devicetree users, such as the Linux kernel.

Zephyr's binding language *semantics* can support Zephyr-specific attributes, but shall not express Zephyr-specific relationships.

Examples

- Zephyr's devicetree source parser, *dtlib.py*, is source-compatible with other tools like dtc in both directions: `dtlib.py` can parse `dtc` output, and `dtc` can parse `dtlib.py` output.
- Zephyr's "extended dtlib" library, `edtlib.py`, shall not include Zephyr-specific features. Its purpose is to provide a higher-level view of the devicetree for common elements like interrupts and buses.

  Only the high-level `gen_defines.py` script, which is built on top of `edtlib.py`, contains Zephyr-specific knowledge and features.

Example remaining work

- Zephyr has a custom *Devicetree bindings* language *syntax*. While Linux's dtschema does not yet meet Zephyr's needs, we should try to follow what it is capable of representing in Zephyr's own bindings.
- Due to inflexibility in the bindings language, Zephyr cannot support the full set of bindings supported by Linux.
- Devicetree source sharing between Zephyr and Linux is not done.

8.11.3 Devicetree bindings

A devicetree on its own is only half the story for describing hardware, as it is a relatively unstructured format. *Devicetree bindings* provide the other half.

A devicetree binding declares requirements on the contents of nodes, and provides semantic information about the contents of valid nodes. Zephyr devicetree bindings are YAML files in a custom format (Zephyr does not use the dt-schema tools used by the Linux kernel).

This page introduces bindings, describes what they do, notes where they are found, and explains their data format.

Note: See the *Bindings index* for reference information on bindings built in to Zephyr.

Introduction

Devicetree nodes are matched to bindings using their *compatible properties*.

During the *Configuration Phase*, the build system tries to match each node in the devicetree to a binding file. When this succeeds, the build system uses the information in the binding file both when validating the node's contents and when generating macros for the node.

A simple example Here is an example devicetree node:

```
/* Node in a DTS file */
bar-device {
     compatible = "foo-company,bar-device";
     num-foos = <3>;
};
```

Here is a minimal binding file which matches the node:

```
# A YAML binding matching the node

compatible: "foo-company,bar-device"

properties:
  num-foos:
    type: int
    required: true
```

The build system matches the `bar-device` node to its YAML binding because the node's `compatible` property matches the binding's `compatible:` line.

What the build system does with bindings The build system uses bindings both to validate devicetree nodes and to convert the devicetree's contents into the generated *devicetree_unfixed.h* header file.

For example, the build system would use the above binding to check that the required `num-foos` property is present in the `bar-device` node, and that its value, `<3>`, has the correct type.

The build system will then generate a macro for the `bar-device` node's `num-foos` property, which will expand to the integer literal `3`. This macro lets you get the value of the property in C code using the API which is discussed later in this guide in *Devicetree access from C/C++*.

For another example, the following node would cause a build error, because it has no `num-foos` property, and this property is marked required in the binding:

```
bad-node {
     compatible = "foo-company,bar-device";
};
```

Other ways nodes are matched to bindings If a node has more than one string in its `compatible` property, the build system looks for compatible bindings in the listed order and uses the first match.

Take this node as an example:

```
baz-device {
     compatible = "foo-company,baz-device", "generic-baz-device";
};
```

The `baz-device` node would get matched to a binding with a `compatible: "generic-baz-device"` line if the build system can't find a binding with a `compatible: "foo-company,baz-device"` line.

Nodes without compatible properties can be matched to bindings associated with their parent nodes. These are called "child bindings". If a node describes hardware on a bus, like I2C or SPI, then the bus type is also taken into account when matching nodes to bindings. (The *Bindings file syntax* section below describes how to write child bindings and bus-specific bindings.)

Some special nodes without `compatible` properties are matched to *Inferred bindings*. For these nodes, the build system generates macros based on the properties in the final devicetree.

Where bindings are located Binding file names usually match their `compatible:` lines. For example, the above example binding would be named `foo-company,bar-device.yaml` by convention.

The build system looks for bindings in `dts/bindings` subdirectories of the following places:

- the zephyr repository
- your *application source directory*
- your *board directory*
- any directories in the *DTS_ROOT* CMake variable
- any *module* that defines a `dts_root` in its *Build settings*

The build system will consider any YAML file in any of these, including in any subdirectories, when matching nodes to bindings. A file is considered YAML if its name ends with `.yaml` or `.yml`.

Warning: The binding files must be located somewhere inside the `dts/bindings` subdirectory of the above places.

For example, if `my-app` is your application directory, then you must place application-specific bindings inside `my-app/dts/bindings`. So `my-app/dts/bindings/serial/my-company,my-serial-port.yaml` would be found, but `my-app/my-company,my-serial-port.yaml` would be ignored.

Bindings file syntax

Zephyr bindings files are YAML files. The top-level value in the file is a mapping. A *simple example* is given above.

The top-level keys in the mapping look like this:

```
# A high level description of the device the binding applies to:
description: |
   This is the Vendomatic company's foo-device.

   Descriptions which span multiple lines (like this) are OK,
   and are encouraged for complex bindings.

   See https://yaml-multiline.info/ for formatting help.
```

(continues on next page)

(continued from previous page)

```
# You can include definitions from other bindings using this syntax:
include: other.yaml

# Used to match nodes to this binding as discussed above:
compatible: "manufacturer,foo-device"

properties:
  # Requirements for and descriptions of the properties that this
  # binding's nodes need to satisfy go here.

child-binding:
  # You can constrain the children of the nodes matching this binding
  # using this key.

# If the node describes bus hardware, like an SPI bus controller
# on an SoC, use 'bus:' to say which one, like this:
bus: spi

# If the node instead appears as a device on a bus, like an external
# SPI memory chip, use 'on-bus:' to say what type of bus, like this.
# Like 'compatible', this key also influences the way nodes match
# bindings.
on-bus: spi

foo-cells:
  # "Specifier" cell names for the 'foo' domain go here; example 'foo'
  # values are 'gpio', 'pwm', and 'dma'. See below for more information.
```

The following sections describe these keys in more detail:

- *Description*
- *Compatible*
- *Properties*
- *Child-binding*
- *Bus*
- *On-bus*
- *Specifier cell names (*-cells)*
- *Include*

The `include:` key usually appears early in the binding file, but it is documented last here because you need to know how the other keys work before understanding `include:`.

Description A free-form description of node hardware goes here. You can put links to datasheets or example nodes or properties as well.

Compatible This key is used to match nodes to this binding as described above. It should look like this in a binding file:

```
# Note the comma-separated vendor prefix and device name
compatible: "manufacturer,device"
```

This devicetree node would match the above binding:

```
device {
     compatible = "manufacturer,device";
};
```

Assuming no binding has `compatible: "manufacturer,device-v2"`, it would also match this node:

```
device-2 {
    compatible = "manufacturer,device-v2", "manufacturer,device";
};
```

Each node's `compatible` property is tried in order. The first matching binding is used. The *on-bus:* key can be used to refine the search.

If more than one binding for a compatible is found, an error is raised.

The `manufacturer` prefix identifies the device vendor. See dts/bindings/vendor-prefixes.txt for a list of accepted vendor prefixes. The `device` part is usually from the datasheet.

Some bindings apply to a generic class of devices which do not have a specific vendor. In these cases, there is no vendor prefix. One example is the `gpio-leds` compatible which is commonly used to describe board LEDs connected to GPIOs.

If more than one binding for a compatible is found, an error is raised.

Properties The `properties:` key describes the properties that nodes which match the binding can contain.

For example, a binding for a UART peripheral might look something like this:

```
compatible: "manufacturer,serial"

properties:
  reg:
    type: array
    description: UART peripheral MMIO register space
    required: true
  current-speed:
    type: int
    description: current baud rate
    required: true
  label:
    type: string
    description: human-readable name
    required: false
```

The properties in the following node would be validated by the above binding:

```
my-serial@deadbeef {
     compatible = "manufacturer,serial";
     reg = <0xdeadbeef 0x1000>;
     current-speed = <115200>;
     label = "UART_0";
};
```

This is used to check that required properties appear, and to control the format of output generated for them.

Except for some special properties, like `reg`, whose meaning is defined by the devicetree specification itself, only properties listed in the `properties:` key will have generated macros.

Example property definitions Here are some more examples.

```
properties:
    # Describes a property like 'current-speed = <115200>;'. We pretend that
    # it's obligatory for the example node and set 'required: true'.
    current-speed:
        type: int
        required: true
        description: Initial baud rate for bar-device

    # Describes an optional property like 'keys = "foo", "bar";'
    keys:
        type: string-array
        required: false
        description: Keys for bar-device

    # Describes an optional property like 'maximum-speed = "full-speed";'
    # the enum specifies known values that the string property may take
    maximum-speed:
        type: string
        required: false
        description: Configures USB controllers to work up to a specific speed.
        enum:
           - "low-speed"
           - "full-speed"
           - "high-speed"
           - "super-speed"

    # Describes an optional property like 'resolution = <16>;'
    # the enum specifies known values that the int property may take
    resolution:
      type: int
      required: false
      enum:
       - 8
       - 16
       - 24
       - 32

    # Describes a required property '#address-cells = <1>';  the const
    # specifies that the value for the property is expected to be the value 1
    "#address-cells":
        type: int
        required: true
        const: 1

    int-with-default:
        type: int
        required: false
        default: 123
        description: Value for int register, default is power-up configuration.

    array-with-default:
        type: array
        required: false
        default: [1, 2, 3] # Same as 'array-with-default = <1 2 3>'

    string-with-default:
```

(continues on next page)

(continued from previous page)

```
        type: string
        required: false
        default: "foo"

    string-array-with-default:
        type: string-array
        required: false
        default: ["foo", "bar"] # Same as 'string-array-with-default = "foo", "bar"'

    uint8-array-with-default:
        type: uint8-array
        required: false
        default: [0x12, 0x34] # Same as 'uint8-array-with-default = [12 34]'
```

Property entry syntax As shown by the above examples, each property entry in a binding looks like this:

```
<property name>:
  required: <true | false>
  type: <string | int | boolean | array | uint8-array | string-array |
         phandle | phandles | phandle-array | path | compound>
  deprecated: <true | false>
  default: <default>
  description: <description of the property>
  enum:
    - <item1>
    - <item2>
    ...
    - <itemN>
  const: <string | int>
```

Required properties If a node matches a binding but is missing any property which the binding defines with `required: true`, the build fails.

Property types The type of a property constrains its values. The following types are available. See *Writing property values* for more details about writing values of each type in a DTS file.

Type	Description	Example in DTS
`string`	exactly one string	`label = "UART_0";`
`int`	exactly one 32-bit value (cell)	`current-speed = <115200>;`
`boolean`	flags that don't take a value when true, and are absent if false	`hw-flow-control;`
`array`	zero or more 32-bit values (cells)	`offsets = <0x100 0x200 0x300>;`
`uint8-array`	zero or more bytes, in hex ('bytestring' in the Devicetree specification)	`local-mac-address = [de ad be ef 12 34];`
`string-array`	zero or more strings	`dma-names = "tx", "rx";`
`phandle`	exactly one phandle	`interrupt-parent = <&gic>;`
`phandles`	zero or more phandles	`pinctrl-0 = <&usart2_tx_pd5 &usart2_rx_pd6>;`
`phandle-array`	a list of phandles and 32-bit cells (usually specifiers)	`dmas = <&dma0 2>, <&dma0 3>;`
`path`	a path to a node as a phandle path reference or path string	`zephyr,bt-c2h-uart = &uart0;` or `foo = "/path/to/some/node";`
`compound`	a catch-all for more complex types (no macros will be generated)	`foo = <&label>, [01 02];`

Deprecated properties A property with `deprecated: true` indicates to both the user and the tooling that the property is meant to be phased out.

The tooling will report a warning if the devicetree includes the property that is flagged as deprecated. (This warning is upgraded to an error in the *Test Runner (Twister)* for upstream pull requests.)

Default values for properties The optional `default:` setting gives a value that will be used if the property is missing from the devicetree node.

For example, with this binding fragment:

```
properties:
  foo:
    type: int
    default: 3
```

If property `foo` is missing in a matching node, then the output will be as if `foo = <3>;` had appeared in the DTS (except YAML data types are used for the default value).

Note that it only makes sense to combine `default:` with `required: false`. Combining it with `required: true` will raise an error.

There is a risk in using `default:` when the value in the binding may be incorrect for a particular board or hardware configuration. For example, defaulting the capacity of the connected power cell in a charging IC binding is likely to be incorrect. For such properties it's better to make the property `required: true`, forcing the devicetree maintainer into an explicit and witting choice.

Driver developers should use their best judgment as to whether a value can be safely defaulted. Candidates for default values include:

- delays that would be different only under unusual conditions (such as intervening hardware)
- configuration for devices that have a standard initial configuration (such as a USB audio headset)
- defaults which match the vendor-specified power-on reset value (as long as they are independent from other properties)

Power-on reset values may be used for defaults as long as they're independent. If changing one property would require changing another to create a consistent configuration, then those properties should be made required.

In any case where `default:` is used, the property documentation should explain why the value was selected and any conditions that would make it necessary to provide a different value. (This is mandatory for built-in bindings.)

See *Example property definitions* for examples. Putting `default:` on any property type besides those used in the examples will raise an error.

Enum values The `enum:` line is followed by a list of values the property may contain. If a property value in DTS is not in the `enum:` list in the binding, an error is raised. See *Example property definitions* for examples.

Const This specifies a constant value the property must take. It is mainly useful for constraining the values of common properties for a particular piece of hardware.

Child-binding `child-binding` can be used when a node has children that all share the same properties. Each child gets the contents of `child-binding` as its binding, though an explicit `compatible = ...` on the child node takes precedence, if a binding is found for it.

Consider a binding for a PWM LED node like this one, where the child nodes are required to have a `pwms` property:

```
pwmleds {
        compatible = "pwm-leds";

        red_pwm_led {
                pwms = <&pwm3 4 15625000>;
        };
        green_pwm_led {
                pwms = <&pwm3 0 15625000>;
        };
        /* ... */
};
```

The binding would look like this:

```
compatible: "pwm-leds"

child-binding:
  description: LED that uses PWM

  properties:
    pwms:
      type: phandle-array
      required: true
```

`child-binding` also works recursively. For example, this binding:

```
compatible: foo

child-binding:
  child-binding:
    properties:
      my-property:
```

(continues on next page)

(continued from previous page)

```
        type: int
        required: true
```

will apply to the `grandchild` node in this DTS:

```
parent {
        compatible = "foo";
        child {
                grandchild {
                        my-property = <123>;
                };
        };
};
```

Bus If the node is a bus controller, use `bus:` in the binding to say what type of bus. For example, a binding for a SPI peripheral on an SoC would look like this:

```
compatible: "manufacturer,spi-peripheral"
bus: spi
# ...
```

The presence of this key in the binding informs the build system that the children of any node matching this binding appear on this type of bus.

This in turn influences the way `on-bus:` is used to match bindings for the child nodes.

On-bus If the node appears as a device on a bus, use `on-bus:` in the binding to say what type of bus.

For example, a binding for an external SPI memory chip should include this line:

```
on-bus: spi
```

And a binding for an I2C based temperature sensor should include this line:

```
on-bus: i2c
```

When looking for a binding for a node, the build system checks if the binding for the parent node contains `bus: <bus type>`. If it does, then only bindings with a matching `on-bus: <bus type>` and bindings without an explicit `on-bus` are considered. Bindings with an explicit `on-bus: <bus type>` are searched for first, before bindings without an explicit `on-bus`. The search repeats for each item in the node's `compatible` property, in order.

This feature allows the same device to have different bindings depending on what bus it appears on. For example, consider a sensor device with compatible `manufacturer,sensor` which can be used via either I2C or SPI.

The sensor node may therefore appear in the devicetree as a child node of either an SPI or an I2C controller, like this:

```
spi-bus@0 {
   /* ... some compatible with 'bus: spi', etc. ... */

   sensor@0 {
       compatible = "manufacturer,sensor";
       reg = <0>;
       /* ... */
   };
```

(continues on next page)

(continued from previous page)

```
};

i2c-bus@0 {
   /* ... some compatible with 'bus: i2c', etc. ... */

   sensor@79 {
       compatible = "manufacturer,sensor";
       reg = <79>;
       /* ... */
   };
};
```

You can write two separate binding files which match these individual sensor nodes, even though they have the same compatible:

```
# manufacturer,sensor-spi.yaml, which matches sensor@0 on the SPI bus:
compatible: "manufacturer,sensor"
on-bus: spi

# manufacturer,sensor-i2c.yaml, which matches sensor@79 on the I2C bus:
compatible: "manufacturer,sensor"
properties:
  uses-clock-stretching:
    type: boolean
    required: false
on-bus: i2c
```

Only `sensor@79` can have a `use-clock-stretching` property. The bus-sensitive logic ignores `manufacturer,sensor-i2c.yaml` when searching for a binding for `sensor@0`.

Specifier cell names (*-cells) Specifier cells are usually used with `phandle-array` type properties briefly introduced above.

To understand the purpose of `*-cells`, assume that some node has the following `pwms` property with type `phandle-array`:

```
my-device {
     pwms = <&pwm0 1 2>, <&pwm3 4>;
};
```

The tooling strips the final `s` from the property name of such properties, resulting in `pwm`. Then the value of the `#pwm-cells` property is looked up in each of the PWM controller nodes `pwm0` and `pwm3`, like so:

```
pwm0: pwm@0 {
     compatible = "foo,pwm";
     #pwm-cells = <2>;
};

pwm3: pwm@3 {
     compatible = "bar,pwm";
     #pwm-cells = <1>;
};
```

The `&pwm0 1 2` part of the property value has two cells, `1` and `2`, which matches `#pwm-cells = <2>;`, so these cells are considered the *specifier* associated with `pwm0` in the phandle array.

Similarly, the cell `4` is the specifier associated with `pwm3`.

The number of PWM cells in the specifiers in `pwms` must match the `#pwm-cells` values, as shown above. If there is a mismatch, an error is raised. For example, this node would result in an error:

```
my-bad-device {
     /* wrong: 2 cells given in the specifier, but #pwm-cells is 1 in pwm3. */
     pwms = <&pwm3 5 6>;
};
```

The binding for each PWM controller must also have a `*-cells` key, in this case `pwm-cells`, giving names to the cells in each specifier:

```
# foo,pwm.yaml
compatible: "foo,pwm"
...
pwm-cells:
  - channel
  - period

# bar,pwm.yaml
compatible: "bar,pwm"
...
pwm-cells:
  - period
```

A `*-names` (e.g. `pwm-names`) property can appear on the node as well, giving a name to each entry.

This allows the cells in the specifiers to be accessed by name, e.g. using APIs like *DT_PWMS_CHANNEL_BY_NAME*.

Because other property names are derived from the name of the property by removing the final `s`, the property name must end in `s`. An error is raised if it doesn't.

`*-gpios` properties are special-cased so that e.g. `foo-gpios` resolves to `#gpio-cells` rather than `#foo-gpio-cells`.

If the specifier is empty (e.g. `#clock-cells = <0>`), then `*-cells` can either be omitted (recommended) or set to an empty array. Note that an empty array is specified as e.g. `clock-cells: []` in YAML.

All `phandle-array` type properties support mapping through `*-map` properties, e.g. `gpio-map`, as defined by the Devicetree specification.

Include Bindings can include other files, which can be used to share common property definitions between bindings. Use the `include:` key for this. Its value is either a string or a list.

In the simplest case, you can include another file by giving its name as a string, like this:

```
include: foo.yaml
```

If any file named `foo.yaml` is found (see *Where bindings are located* for the search process), it will be included into this binding.

Included files are merged into bindings with a simple recursive dictionary merge. The build system will check that the resulting merged binding is well-formed.

It is an error if a key appears with a different value in a binding and in a file it includes, with one exception: a binding can have `required: true` for a *property definition* for which the included file has `required: false`. The `required: true` takes precedence, allowing bindings to strengthen requirements from included files.

Note that weakening requirements by having `required: false` where the included file has `required: true` is an error. This is meant to keep the organization clean.

The file base.yaml contains definitions for many common properties. When writing a new binding, it is a good idea to check if `base.yaml` already defines some of the needed properties, and include it if it does.

Note that you can make a property defined in base.yaml obligatory like this, taking *reg* as an example:

```
reg:
  required: true
```

This relies on the dictionary merge to fill in the other keys for `reg`, like `type`.

To include multiple files, you can use a list of strings:

```
include:
  - foo.yaml
  - bar.yaml
```

This includes the files `foo.yaml` and `bar.yaml`. (You can write this list in a single line of YAML as `include: [foo.yaml, bar.yaml]`.)

When including multiple files, any overlapping `required` keys on properties in the included files are ORed together. This makes sure that a `required: true` is always respected.

In some cases, you may want to include some property definitions from a file, but not all of them. In this case, `include:` should be a list, and you can filter out just the definitions you want by putting a mapping in the list, like this:

```
include:
  - name: foo.yaml
    property-allowlist:
      - i-want-this-one
      - and-this-one
  - name: bar.yaml
    property-blocklist:
      - do-not-include-this-one
      - or-this-one
```

Each map element must have a `name` key which is the filename to include, and may have `property-allowlist` and `property-blocklist` keys that filter which properties are included.

You cannot have a single map element with both `property-allowlist` and `property-blocklist` keys. A map element with neither `property-allowlist` nor `property-blocklist` is valid; no additional filtering is done.

You can freely intermix strings and mappings in a single `include:` list:

```
include:
  - foo.yaml
  - name: bar.yaml
    property-blocklist:
      - do-not-include-this-one
      - or-this-one
```

Finally, you can filter from a child binding like this:

```
include:
  - name: bar.yaml
    child-binding:
      property-allowlist:
        - child-prop-to-allow
```

Inferred bindings

Zephyr's devicetree scripts can "infer" a binding for the special `/zephyr,user` node based on the values observed in its properties.

This node matches a binding which is dynamically created by the build system based on the values of its properties in the final devicetree. It does not have a `compatible` property.

This node is meant for sample code and applications. The devicetree API provides it as a convenient container when only a few simple properties are needed, such as storing a hardware-dependent value, phandle(s), or GPIO pin.

For example, with this DTS fragment:

```
#include <dt-bindings/gpio/gpio.h>

/ {
     zephyr,user {
             boolean;
             bytes = [81 82 83];
             number = <23>;
             numbers = <1>, <2>, <3>;
             string = "text";
             strings = "a", "b", "c";

             handle = <&gpio0>;
             handles = <&gpio0>, <&gpio1>;
             signal-gpios = <&gpio0 1 GPIO_ACTIVE_HIGH>;
     };
};
```

You can get the simple values like this:

```
#define ZEPHYR_USER_NODE DT_PATH(zephyr_user)

DT_PROP(ZEPHYR_USER_NODE, boolean) // 1
DT_PROP(ZEPHYR_USER_NODE, bytes)   // {0x81, 0x82, 0x83}
DT_PROP(ZEPHYR_USER_NODE, number)  // 23
DT_PROP(ZEPHYR_USER_NODE, numbers) // {1, 2, 3}
DT_PROP(ZEPHYR_USER_NODE, string)  // "text"
DT_PROP(ZEPHYR_USER_NODE, strings) // {"a", "b", "c"}
```

You can convert the phandles in the `handle` and `handles` properties to device pointers like this:

```
/*
 * Same thing as:
 *
 * ... my_dev = DEVICE_DT_GET(DT_NODELABEL(gpio0));
 */
const struct device *my_device =
     DEVICE_DT_GET(DT_PROP(ZEPHYR_USER_NODE, handle));

#define PHANDLE_TO_DEVICE(node_id, prop, idx) \
     DEVICE_DT_GET(DT_PHANDLE_BY_IDX(node_id, prop, idx)),

/*
 * Same thing as:
 *
 * ... *my_devices[] = {
 *         DEVICE_DT_GET(DT_NODELABEL(gpio0)),
```

(continues on next page)

(continued from previous page)

```
 *         DEVICE_DT_GET(DT_NODELABEL(gpio1)),
 * };
 */
const struct device *my_devices[] = {
    DT_FOREACH_PROP_ELEM(ZEPHYR_USER_NODE, handles, PHANDLE_TO_DEVICE)
};
```

And you can convert the pin defined in `signal-gpios` to a `struct gpio_dt_spec`, then use it like this:

```
#include <drivers/gpio.h>

#define ZEPHYR_USER_NODE DT_PATH(zephyr_user)

const struct gpio_dt_spec signal =
        GPIO_DT_SPEC_GET(ZEPHYR_USER_NODE, signal_gpios);

/* Configure the pin */
gpio_pin_configure_dt(&signal, GPIO_OUTPUT_INACTIVE);

/* Set the pin to its active level */
gpio_pin_set(signal.port, signal.pin, 1);
```

(See *gpio_dt_spec*, *GPIO_DT_SPEC_GET*, and *gpio_pin_configure_dt()* for details on these APIs.)

8.11.4 Devicetree access from C/C++

This guide describes Zephyr's `<devicetree.h>` API for reading the devicetree from C source files. It assumes you're familiar with the concepts in *Introduction to devicetree* and *Devicetree bindings*. See *Devicetree* for reference material.

A note for Linux developers

Linux developers familiar with devicetree should be warned that the API described here differs significantly from how devicetree is used on Linux.

Instead of generating a C header with all the devicetree data which is then abstracted behind a macro API, the Linux kernel would instead read the devicetree data structure in its binary form. The binary representation is parsed at runtime, for example to load and initialize device drivers.

Zephyr does not work this way because the size of the devicetree binary and associated handling code would be too large to fit comfortably on the relatively constrained devices Zephyr supports.

Node identifiers

To get information about a particular devicetree node, you need a *node identifier* for it. This is a just a C macro that refers to the node.

These are the main ways to get a node identifier:

By path Use *DT_PATH()* along with the node's full path in the devicetree, starting from the root node. This is mostly useful if you happen to know the exact node you're looking for.

By node label Use *DT_NODELABEL()* to get a node identifier from a *node label*. Node labels are often provided by SoC `.dtsi` to give nodes names that match the SoC datasheet, like `i2c1`, `spi2`, etc.

By alias Use *DT_ALIAS()* to get a node identifier for a property of the special `/aliases` node. This is sometmes done by applications (like blinky, which uses the `led0` alias) that need to refer to *some* device of a particular type ("the board's user LED") but don't care which one is used.

By instance number This is done primarily by device drivers, as instance numbers are a way to refer to individual nodes based on a matching compatible. Get these with *DT_INST()*, but be careful doing so. See below.

By chosen node Use *DT_CHOSEN()* to get a node identifier for `/chosen` node properties.

By parent/child Use *DT_PARENT()* and *DT_CHILD()* to get a node identifier for a parent or child node, starting from a node identifier you already have.

Two node identifiers which refer to the same node are identical and can be used interchangeably.

Here's a DTS fragment for some imaginary hardware we'll return to throughout this file for examples:

```
/dts-v1/;

/ {

        aliases {
                sensor-controller = &i2c1;
        };

        soc {
                i2c1: i2c@40002000 {
                        compatible = "vnd,soc-i2c";
                        label = "I2C_1";
                        reg = <0x40002000 0x1000>;
                        status = "okay";
                        clock-frequency = < 100000 >;
                };
        };
};
```

Here are a few ways to get node identifiers for the `i2c@40002000` node:

- `DT_PATH(soc, i2c_40002000)`
- `DT_NODELABEL(i2c1)`
- `DT_ALIAS(sensor_controller)`
- `DT_INST(x, vnd_soc_i2c)` for some unknown number `x`. See the *DT_INST()* documentation for details.

Important: Non-alphanumeric characters like dash (`-`) and the at sign (`@`) in devicetree names are converted to underscores (`_`). The names in a DTS are also converted to lowercase.

Node identifiers are not values

There is no way to store one in a variable. You cannot write:

```
/* These will give you compiler errors: */

void *i2c_0 = DT_INST(0, vnd_soc_i2c);
unsigned int i2c_1 = DT_INST(1, vnd_soc_i2c);
long my_i2c = DT_NODELABEL(i2c1);
```

If you want something short to save typing, use C macros:

```
/* Use something like this instead: */

#define MY_I2C DT_NODELABEL(i2c1)

#define INST(i) DT_INST(i, vnd_soc_i2c)
#define I2C_0 INST(0)
#define I2C_1 INST(1)
```

Property access

The right API to use to read property values depends on the node and property.

- *Checking properties and values*
- *Simple properties*
- *reg properties*
- *interrupts properties*
- *phandle properties*

Checking properties and values You can use *DT_NODE_HAS_PROP()* to check if a node has a property. For the *example devicetree* above:

```
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), clock_frequency)  /* expands to 1 */
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), not_a_property)   /* expands to 0 */
```

Simple properties Use `DT_PROP(node_id, property)` to read basic integer, boolean, string, numeric array, and string array properties.

For example, to read the `clock-frequency` property's value in the *above example*:

```
DT_PROP(DT_PATH(soc, i2c_40002000), clock_frequency)  /* This is 100000, */
DT_PROP(DT_NODELABEL(i2c1), clock_frequency)          /* and so is this, */
DT_PROP(DT_ALIAS(sensor_controller), clock_frequency) /* and this. */
```

Important: The DTS property `clock-frequency` is spelled `clock_frequency` in C. That is, properties also need special characters converted to underscores. Their names are also forced to lowercase.

Properties with `string` and `boolean` types work the exact same way. The `DT_PROP()` macro expands to a string literal in the case of strings, and the number 0 or 1 in the case of booleans. For example:

```
#define I2C1 DT_NODELABEL(i2c1)

DT_PROP(I2C1, status)  /* expands to the string literal "okay" */
```

Note: Don't use DT_NODE_HAS_PROP() for boolean properties. Use DT_PROP() instead as shown above. It will expand to either 0 or 1 depending on if the property is present or absent.

Properties with type `array`, `uint8-array`, and `string-array` work similarly, except `DT_PROP()` expands to an array initializer in these cases. Here is an example devicetree fragment:

```
foo: foo@1234 {
        a = <1000 2000 3000>; /* array */
        b = [aa bb cc dd];    /* uint8-array */
        c = "bar", "baz";     /* string-array */
};
```

Its properties can be accessed like this:

```
#define FOO DT_NODELABEL(foo)

int a[] = DT_PROP(FOO, a);           /* {1000, 2000, 3000} */
unsigned char b[] = DT_PROP(FOO, b); /* {0xaa, 0xbb, 0xcc, 0xdd} */
char* c[] = DT_PROP(FOO, c);         /* {"foo", "bar"} */
```

You can use *DT_PROP_LEN()* to get logical array lengths in number of elements.

```
size_t a_len = DT_PROP_LEN(FOO, a); /* 3 */
size_t b_len = DT_PROP_LEN(FOO, b); /* 4 */
size_t c_len = DT_PROP_LEN(FOO, c); /* 2 */
```

`DT_PROP_LEN()` cannot be used with the special `reg` or `interrupts` properties. These have alternative macros which are described next.

reg properties See *Important properties* for an introduction to `reg`.

Given a node identifier `node_id`, `DT_NUM_REGS(node_id)` is the total number of register blocks in the node's `reg` property.

You **cannot** read register block addresses and lengths with `DT_PROP(node, reg)`. Instead, if a node only has one register block, use *DT_REG_ADDR()* or *DT_REG_SIZE()*:

- `DT_REG_ADDR(node_id)`: the given node's register block address
- `DT_REG_SIZE(node_id)`: its size

Use *DT_REG_ADDR_BY_IDX()* or *DT_REG_SIZE_BY_IDX()* instead if the node has multiple register blocks:

- `DT_REG_ADDR_BY_IDX(node_id, idx)`: address of register block at index `idx`
- `DT_REG_SIZE_BY_IDX(node_id, idx)`: size of block at index `idx`

The `idx` argument to these must be an integer literal or a macro that expands to one without requiring any arithmetic. In particular, `idx` cannot be a variable. This won't work:

```
/* This will cause a compiler error. */

for (size_t i = 0; i < DT_NUM_REGS(node_id); i++) {
        size_t addr = DT_REG_ADDR_BY_IDX(node_id, i);
}
```

interrupts properties See *Important properties* for a brief introduction to `interrupts`.

Given a node identifier `node_id`, `DT_NUM_IRQS(node_id)` is the total number of interrupt specifiers in the node's `interrupts` property.

The most general purpose API macro for accessing these is *DT_IRQ_BY_IDX()*:

```
DT_IRQ_BY_IDX(node_id, idx, val)
```

Here, `idx` is the logical index into the `interrupts` array, i.e. it is the index of an individual interrupt specifier in the property. The `val` argument is the name of a cell within the interrupt specifier. To use this macro, check the bindings file for the node you are interested in to find the `val` names.

Most Zephyr devicetree bindings have a cell named `irq`, which is the interrupt number. You can use *DT_IRQN()* as a convenient way to get a processed view of this value.

Warning: Here, "processed" reflects Zephyr's devicetree *Scripts and tools*, which change the `irq` number in *zephyr.dts* to handle hardware constraints on some SoCs and in accordance with Zephyr's multilevel interrupt numbering.

This is currently not very well documented, and you'll need to read the scripts' source code and existing drivers for more details if you are writing a device driver.

phandle properties Property values can refer to other nodes using the `&another-node` phandle syntax introduced in *Writing property values*. Properties which contain phandles have type `phandle`, `phandles`, or `phandle-array` in their bindings. We'll call these "phandle properties" for short.

You can convert a phandle to a node identifier using *DT_PHANDLE()*, *DT_PHANDLE_BY_IDX()*, or *DT_PHANDLE_BY_NAME()*, depending on the type of property you are working with.

One common use case for phandle properties is referring to other hardware in the tree. In this case, you usually want to convert the devicetree-level phandle to a Zephyr driver-level *struct device*. See *Get a struct device from a devicetree node* for ways to do that.

Another common use case is accessing specifier values in a phandle array. The general purpose APIs for this are *DT_PHA_BY_IDX()* and *DT_PHA()*. There are also hardware-specific shorthands like *DT_GPIO_CTLR_BY_IDX()*, *DT_GPIO_CTLR()*, *DT_GPIO_LABEL_BY_IDX()*, *DT_GPIO_LABEL()*, *DT_GPIO_PIN_BY_IDX()*, *DT_GPIO_PIN()*, *DT_GPIO_FLAGS_BY_IDX()*, and *DT_GPIO_FLAGS()*.

See *DT_PHA_HAS_CELL_AT_IDX()* and *DT_PROP_HAS_IDX()* for ways to check if a specifier value is present in a phandle property.

Other APIs

Here are pointers to some other available APIs.

- *DT_CHOSEN()*, *DT_HAS_CHOSEN()*: for properties of the special `/chosen` node
- *DT_HAS_COMPAT_STATUS_OKAY()*, *DT_NODE_HAS_COMPAT()*: global- and node-specific tests related to the `compatible` property
- *DT_BUS()*: get a node's bus controller, if there is one
- *DT_ENUM_IDX()*: for properties whose values are among a fixed list of choices
- *Fixed flash partitions*: APIs for managing fixed flash partitions. Also see *Flash map*, which wraps this in a more user-friendly API.

Device driver conveniences

Special purpose macros are available for writing device drivers, which usually rely on *instance identifiers*.

To use these, you must define `DT_DRV_COMPAT` to the `compat` value your driver implements support for. This `compat` value is what you would pass to *DT_INST()*.

If you do that, you can access the properties of individual instances of your compatible with less typing, like this:

```
#include <devicetree.h>

#define DT_DRV_COMPAT my_driver_compat
```

(continues on next page)

(continued from previous page)

```
/* This is same thing as DT_INST(0, my_driver_compat): */
DT_DRV_INST(0)

/*
 * This is the same thing as
 * DT_PROP(DT_INST(0, my_driver_compat), clock_frequency)
 */
DT_INST_PROP(0, clock_frequency)
```

See *Instance-based APIs* for a generic API reference.

Hardware specific APIs

Convenience macros built on top of the above APIs are also defined to help readability for hardware specific code. See *Hardware specific APIs* for details.

Generated macros

While the `devicetree.h` API is not generated, it does rely on a generated C header which is put into every application build directory: *devicetree_unfixed.h*. This file contains macros with devicetree data.

These macros have tricky naming conventions which the *Devicetree API* API abstracts away. They should be considered an implementation detail, but it's useful to understand them since they will frequently be seen in compiler error messages.

This section contains an Augmented Backus-Naur Form grammar for these generated macros, with examples and more details in comments. See RFC 7405 (which extends RFC 5234) for a syntax specification.

```
; An RFC 7405 ABNF grammar for devicetree macros.
;
; This does *not* cover macros pulled out of DT via Kconfig,
; like CONFIG_SRAM_BASE_ADDRESS, etc. It only describes the
; ones that start with DT_ and are directly generated, not
; defined in a dts_fixup.h file.

; ------------------------------------------------------------------------
; dt-macro: the top level nonterminal for a devicetree macro
;
; A dt-macro starts with uppercase "DT_", and is one of:
;
; - a <node-macro>, generated for a particular node
; - some <other-macro>, a catch-all for other types of macros
dt-macro = node-macro / other-macro

; ------------------------------------------------------------------------
; node-macro: a macro related to a node

; A macro about a property value
node-macro =  property-macro
; A macro about the pinctrl properties in a node.
node-macro =/ pinctrl-macro
; EXISTS macro: node exists in the devicetree
node-macro =/ %s"DT_N" path-id %s"_EXISTS"
; Bus macros: the plain BUS is a way to access a node's bus controller.
; The additional dt-name suffix is added to match that node's bus type;
; the dt-name in this case is something like "spi" or "i2c".
```

(continues on next page)

(continued from previous page)

```
node-macro =/ %s"DT_N" path-id %s"_BUS" ["_" dt-name]
; The reg property is special and has its own macros.
node-macro =/ %s"DT_N" path-id %s"_REG_NUM"
node-macro =/ %s"DT_N" path-id %s"_REG_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_REG_IDX_" DIGIT
              %s"_VAL_" ( %s"ADDRESS" / %s"SIZE")
node-macro =/ %s"DT_N" path-id %s"_REG_NAME_" dt-name
              %s"_VAL_" ( %s"ADDRESS" / %s"SIZE")
; The interrupts property is also special.
node-macro =/ %s"DT_N" path-id %s"_IRQ_NUM"
node-macro =/ %s"DT_N" path-id %s"_IRQ_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_IRQ_IDX_" DIGIT
              %s"_VAL_" dt-name [ %s"_EXISTS" ]
node-macro =/ %s"DT_N" path-id %s"_IRQ_NAME_" dt-name
              %s"_VAL_" dt-name [ %s"_EXISTS" ]
; Subnodes of the fixed-partitions compatible get macros which contain
; a unique ordinal value for each partition
node-macro =/ %s"DT_N" path-id %s"_PARTITION_ID" DIGIT
; Macros are generated for each of a node's compatibles;
; dt-name in this case is something like "vnd_device".
node-macro =/ %s"DT_N" path-id %s"_COMPAT_MATCHES_" dt-name
; Every non-root node gets one of these macros, which expands to the node
; identifier for that node's parent in the devicetree.
node-macro =/ %s"DT_N" path-id %s"_PARENT"
; These are used internally by DT_FOREACH_CHILD, which iterates over
; each child node.
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_VARGS"
; These are used internally by DT_FOREACH_CHILD_STATUS_OKAY, which iterates
; over each child node with status "okay".
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY_VARGS"
; The node's status macro; dt-name in this case is something like "okay"
; or "disabled".
node-macro =/ %s"DT_N" path-id %s"_STATUS_" dt-name
; The node's dependency ordinal. This is a non-negative integer
; value that is used to represent dependency information.
node-macro =/ %s"DT_N" path-id %s"_ORD"
; The node's path, as a string literal
node-macro =/ %s"DT_N" path-id %s"_PATH"
; The node's name@unit-addr, as a string literal
node-macro =/ %s"DT_N" path-id %s"_FULL_NAME"
; The dependency ordinals of a node's requirements (direct dependencies).
node-macro =/ %s"DT_N" path-id %s"_REQUIRES_ORDS"
; The dependency ordinals of a node supports (reverse direct dependencies).
node-macro =/ %s"DT_N" path-id %s"_SUPPORTS_ORDS"

; ------------------------------------------------------------------
; pinctrl-macro: a macro related to the pinctrl properties in a node
;
; These are a bit of a special case because they kind of form an array,
; but the array indexes correspond to pinctrl-DIGIT properties in a node.
;
; So they're related to a node, but not just one property within the node.
;
; The following examples assume something like this:
```

(continues on next page)

(continued from previous page)

```
;
;      foo {
;              pinctrl-0 = <&bar>;
;              pinctrl-1 = <&baz>;
;              pinctrl-names = "default", "sleep";
;      };

; Total number of pinctrl-DIGIT properties in the node. May be zero.
;
;   #define DT_N_<node path>_PINCTRL_NUM 2
pinctrl-macro = %s"DT_N" path-id %s"_PINCTRL_NUM"
; A given pinctrl-DIGIT property exists.
;
;     #define DT_N_<node path>_PINCTRL_IDX_0_EXISTS 1
;     #define DT_N_<node path>_PINCTRL_IDX_1_EXISTS 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_IDX_" DIGIT %s"_EXISTS"
; A given pinctrl property name exists.
;
;     #define DT_N_<node path>_PINCTRL_NAME_default_EXISTS 1
;     #define DT_N_<node path>_PINCTRL_NAME_sleep_EXISTS 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_EXISTS"
; The corresponding index number of a named pinctrl property.
;
;     #define DT_N_<node path>_PINCTRL_NAME_default_IDX 0
;     #define DT_N_<node path>_PINCTRL_NAME_sleep_IDX 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_IDX"
; The node identifier for the phandle in a named pinctrl property.
;
;    #define DT_N_<node path>_PINCTRL_NAME_default_IDX_0_PH <node id for 'bar'>
;
; There's no need for a separate macro for access by index: that's
; covered by property-macro. We only need this because the map from
; names to properties is implicit in the structure of the DT.
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_IDX_" DIGIT %s"_PH"

; ------------------------------------------------------------------------
; property-macro: a macro related to a node property
;
; These combine a node identifier with a "lowercase-and-underscores form"
; property name. The value expands to something related to the property's
; value.
;
; The optional prop-suf suffix is when there's some specialized
; subvalue that deserves its own macro, like the macros for an array
; property's individual elements
;
; The "plain vanilla" macro for a property's value, with no prop-suf,
; looks like this:
;
;   DT_N_<node path>_P_<property name>
;
; Components:
;
; - path-id: node's devicetree path converted to a C token
; - prop-id: node's property name converted to a C token
; - prop-suf: an optional property-specific suffix
```

(continues on next page)

(continued from previous page)

```
property-macro =  %s"DT_N" path-id %s"_P_" prop-id [prop-suf]

; ------------------------------------------------------------------------
; path-id: a node's path-based macro identifier
;
; This in "lowercase-and-underscores" form. I.e. it is
; the node's devicetree path converted to a C token by changing:
;
; - each slash (/) to _S_
; - all letters to lowercase
; - non-alphanumerics characters to underscores
;
; For example, the leaf node "bar-BAZ" in this devicetree:
;
;   / {
;           foo@123 {
;                   bar-BAZ {};
;           };
;   };
;
; has path-id "_S_foo_123_S_bar_baz".
path-id = 1*( %s"_S_" dt-name )

; ------------------------------------------------------------------------
; prop-id: a property identifier
;
; A property name converted to a C token by changing:
;
; - all letters to lowercase
; - non-alphanumeric characters to underscores
;
; Example node:
;
;   chosen {
;       zephyr,console = &uart1;
;       WHY,AM_I_SHOUTING = "unclear";
;   };
;
; The 'zephyr,console' property has prop-id 'zephyr_console'.
; 'WHY,AM_I_SHOUTING' has prop-id 'why_am_i_shouting'.
prop-id = dt-name

; ------------------------------------------------------------------------
; prop-suf: a property-specific macro suffix
;
; Extra macros are generated for properties:
;
; - that are special to the specification ("reg", "interrupts", etc.)
; - with array types (uint8-array, phandle-array, etc.)
; - with "enum:" in their bindings
; - that have zephyr device API specific macros for phandle-arrays
; - related to phandle specifier names ("foo-names")
;
; Here are some examples:
;
; - _EXISTS: property, index or name existence flag
```

(continues on next page)

(continued from previous page)

```
; - _SIZE: logical property length
; - _IDX_<i>: values of individual array elements
; - _IDX_<DIGIT>_VAL_<dt-name>: values of individual specifier
;   cells within a phandle array
; - _ADDR_<i>: for reg properties, the i-th register block address
; - _LEN_<i>: for reg properties, the i-th register block length
;
; The different cases are not exhaustively documented here to avoid
; this file going stale. Please see devicetree.h if you need to know
; the details.
prop-suf = 1*( "_" gen-name ["_" dt-name] )

; ---------------------------------------------------------------------
; other-macro: grab bag for everything that isn't a node-macro.

; See examples below.
other-macro =  %s"DT_N_" alternate-id
; Total count of enabled instances of a compatible.
other-macro =/ %s"DT_N_INST_" dt-name %s"_NUM_OKAY"
; These are used internally by DT_FOREACH_STATUS_OKAY,
; which iterates over each enabled node of a compatible.
other-macro =/ %s"DT_FOREACH_OKAY_" dt-name
other-macro =/ %s"DT_FOREACH_OKAY_VARGS_" dt-name
; These are used internally by DT_INST_FOREACH_STATUS_OKAY,
; which iterates over each enabled instance of a compatible.
other-macro =/ %s"DT_FOREACH_OKAY_INST_" dt-name
other-macro =/ %s"DT_FOREACH_OKAY_INST_VARGS_" dt-name
; E.g.: #define DT_CHOSEN_zephyr_flash
other-macro =/ %s"DT_CHOSEN_" dt-name
; Declares that a compatible has at least one node on a bus.
; Example:
;
;   #define DT_COMPAT_vnd_dev_BUS_spi 1
other-macro =/ %s"DT_COMPAT_" dt-name %s"_BUS_" dt-name
; Declares that a compatible has at least one status "okay" node.
; Example:
;
;   #define DT_COMPAT_HAS_OKAY_vnd_dev 1
other-macro =/ %s"DT_COMPAT_HAS_OKAY_" dt-name
; Currently used to allow mapping a lowercase-and-underscores "label"
; property to a fixed-partitions node. See the flash map API docs
; for an example.
other-macro =/ %s"DT_COMPAT_" dt-name %s"_LABEL_" dt-name

; ---------------------------------------------------------------------
; alternate-id: another way to specify a node besides a path-id
;
; Example devicetree:
;
;   / {
;           aliases {
;                   dev = &dev_1;
;           };
;
;           soc {
;               dev_1: device@123 {
```

(continues on next page)

(continued from previous page)

```
;                   compatible = "vnd,device";
;               };
;           };
;   };
;
; Node device@123 has these alternate-id values:
;
; - ALIAS_dev
; - NODELABEL_dev_1
; - INST_0_vnd_device
;
; The full alternate-id macros are:
;
;   #define DT_N_INST_0_vnd_device     DT_N_S_soc_S_device_123
;   #define DT_N_ALIAS_dev             DT_N_S_soc_S_device_123
;   #define DT_N_NODELABEL_dev_1       DT_N_S_soc_S_device_123
;
; These mainly exist to allow pasting an alternate-id macro onto a
; "_P_<prop-id>" to access node properties given a node's alias, etc.
;
; Notice that "inst"-type IDs have a leading instance identifier,
; which is generated by the devicetree scripts. The other types of
; alternate-id begin immediately with names taken from the devicetree.
alternate-id =  ( %s"ALIAS" / %s"NODELABEL" ) dt-name
alternate-id =/ %s"INST_" 1*DIGIT "_" dt-name

; ------------------------------------------------------------------------
; miscellaneous helper definitions

; A dt-name is one or more:
; - lowercase ASCII letters (a-z)
; - numbers (0-9)
; - underscores ("_")
;
; They are the result of converting names or combinations of names
; from devicetree to a valid component of a C identifier by
; lowercasing letters (in practice, this is a no-op) and converting
; non-alphanumeric characters to underscores.
;
; You'll see these referred to as "lowercase-and-underscores" forms of
; various devicetree identifiers throughout the documentation.
dt-name = 1*( lower / DIGIT / "_" )

; gen-name is used as a stand-in for a component of a generated macro
; name which does not come from devicetree (dt-name covers that case).
;
; - uppercase ASCII letters (a-z)
; - numbers (0-9)
; - underscores ("_")
gen-name = upper 1*( upper / DIGIT / "_" )

; "lowercase ASCII letter" turns out to be pretty annoying to specify
; in RFC-7405 syntax.
;
; This is just ASCII letters a (0x61) through z (0x7a).
lower = %x61-7A
```

(continues on next page)

(continued from previous page)

```
; "uppercase ASCII letter" in RFC-7405 syntax
upper = %x41-5A
```

8.11.5 Devicetree HOWTOs

This page has step-by-step advice for getting things done with devicetree.

Tip: See *Troubleshooting devicetree* for troubleshooting advice.

Get your devicetree and generated header

A board's devicetree (*BOARD.dts*) pulls in common node definitions via `#include` preprocessor directives. This at least includes the SoC's `.dtsi`. One way to figure out the devicetree's contents is by opening these files, e.g. by looking in `dts/<ARCH>/<vendor>/<soc>.dtsi`, but this can be time consuming.

If you just want to see the "final" devicetree for your board, build an application and open the `zephyr.dts` file in the build directory.

Tip: You can build hello_world to see the "base" devicetree for your board without any additional changes from *overlay files*.

For example, using the qemu_cortex_m3 board to build hello_world:

```
# --cmake-only here just forces CMake to run, skipping the
# build process to save time.
west build -b qemu_cortex_m3 -s samples/hello_world --cmake-only
```

You can change `qemu_cortex_m3` to match your board.

CMake prints the input and output file locations like this:

```
-- Found BOARD.dts: .../zephyr/boards/arm/qemu_cortex_m3/qemu_cortex_m3.dts
-- Generated zephyr.dts: .../zephyr/build/zephyr/zephyr.dts
-- Generated devicetree_unfixed.h: .../zephyr/build/zephyr/include/generated/
↪devicetree_unfixed.h
```

The `zephyr.dts` file is the final devicetree in DTS format.

The `devicetree_unfixed.h` file is the corresponding generated header.

See *Input and output files* for details about these files.

Get a struct device from a devicetree node

When writing Zephyr applications, you'll often want to get a driver-level *struct device* corresponding to a devicetree node.

For example, with this devicetree fragment, you might want the struct device for `serial@40002000`:

```
/ {
        soc {
                serial0: serial@40002000 {
```

(continues on next page)

(continued from previous page)

```
                        status = "okay";
                        current-speed = <115200>;
                        /* ... */
                };
        };

        aliases {
                my-serial = &serial0;
        };

        chosen {
                zephyr,console = &serial0;
        };
};
```

Start by making a *node identifier* for the device you are interested in. There are different ways to do this; pick whichever one works best for your requirements. Here are some examples:

```
/* Option 1: by node label */
#define MY_SERIAL DT_NODELABEL(serial0)

/* Option 2: by alias */
#define MY_SERIAL DT_ALIAS(my_serial)

/* Option 3: by chosen node */
#define MY_SERIAL DT_CHOSEN(zephyr_console)

/* Option 4: by path */
#define MY_SERIAL DT_PATH(soc, serial_40002000)
```

Once you have a node identifier there are two ways to proceed. The classic way is to get the `struct device` by combining *DT_LABEL()* with *device_get_binding()*:

```
const struct device *uart_dev = device_get_binding(DT_LABEL(MY_SERIAL));
```

You can then use `uart_dev` with *UART* API functions like *uart_configure()*. Similar code will work for other device types; just make sure you use the correct API for the device.

There's no need to override the `label` property to something else: just make a node identifier and pass it to `DT_LABEL` to get the right string to pass to `device_get_binding()`.

The second way to get a device is to use *DEVICE_DT_GET()*:

```
const struct device *uart_dev = DEVICE_DT_GET(MY_SERIAL);

if (!device_is_ready(uart_dev)) {
        /* Not ready, do not use */
        return -ENODEV;
}
```

This idiom fetches the device pointer at build-time, which is useful when you want to store the device pointer as configuration data. But because the device may not be initialized, or may have failed to initialize, you must verify that the device is ready to be used before passing it to any API functions. (This check is done for you by *device_get_binding()*.)

If you're having trouble, see *Troubleshooting devicetree*. The first thing to check is that the node has `status = "okay"`, like this:

```
#define MY_SERIAL DT_NODELABEL(my_serial)

#if DT_NODE_HAS_STATUS(MY_SERIAL, okay)
const struct device *uart_dev = device_get_binding(DT_LABEL(MY_SERIAL));
#else
#error "Node is disabled"
#endif
```

If you see the `#error` output, make sure to enable the node in your devicetree. If you don't see the `#error` but `uart_dev` is NULL, then there's likely either a Kconfig issue preventing the device driver from creating the device, or the device's initialization function failed.

Find a devicetree binding

Devicetree bindings are YAML files which declare what you can do with the nodes they describe, so it's critical to be able to find them for the nodes you are using.

If you don't have them already, *Get your devicetree and generated header*. To find a node's binding, open the generated header file, which starts with a list of nodes in a block comment:

```
/*
 * [...]
 * Nodes in dependency order (ordinal and path):
 *   0   /
 *   1   /aliases
 *   2   /chosen
 *   3   /flash@0
 *   4   /memory@20000000
 *          (etc.)
 * [...]
 */
```

Make note of the path to the node you want to find, like `/flash@0`. Search for the node's output in the file, which starts with something like this if the node has a matching binding:

```
/*
 * Devicetree node:
 *   /flash@0
 *
 * Binding (compatible = soc-nv-flash):
 *   $ZEPHYR_BASE/dts/bindings/mtd/soc-nv-flash.yaml
 * [...]
 */
```

See *Check for missing bindings* for troubleshooting.

Set devicetree overlays

Devicetree overlays are explained in *Introduction to devicetree*. The CMake variable `DTC_OVERLAY_FILE` contains a space- or semicolon-separated list of overlays. If `DTC_OVERLAY_FILE` specifies multiple files, they are included in that order by the C preprocessor.

Here are some ways to set it:

1. on the cmake build command line (`-DDTC_OVERLAY_FILE="file1.overlay;file2.overlay"`)
2. with the CMake `set()` command in the application `CMakeLists.txt`, before including zephyr's `boilerplate.cmake` file

3. create a `boards/<BOARD>_<revision>.overlay` file in the application folder for the current board revision. This requires that the board supports multiple revisions, see *Multiple board revisions*. The `boards/<BOARD>_<revision>.overlay` file will be merged with `boards/<BOARD>.overlay` if this file also exists.
4. create a `boards/<BOARD>.overlay` file in the application folder, for the current board
5. create a `<BOARD>.overlay` file in the application folder
6. create an `app.overlay` file in the application folder

Here is an example *using west build*. However you set the value, it is saved in the CMake cache between builds.

The *build system* prints all the devicetree overlays it finds in the configuration phase, like this:

```
-- Found devicetree overlay: .../some/file.overlay
```

Use devicetree overlays

See *Set devicetree overlays* for how to add an overlay to the build.

Overlays can override node property values in multiple ways. For example, if your BOARD.dts contains this node:

```
/ {
        soc {
                serial0: serial@40002000 {
                        status = "okay";
                        current-speed = <115200>;
                        /* ... */
                };
        };
};
```

These are equivalent ways to override the `current-speed` value in an overlay:

```
/* Option 1 */
&serial0 {
     current-speed = <9600>;
};

/* Option 2 */
&{/soc/serial@40002000} {
     current-speed = <9600>;
};
```

We'll use the `&serial0` style for the rest of these examples.

You can add aliases to your devicetree using overlays: an alias is just a property of the `/aliases` node. For example:

```
/ {
     aliases {
             my-serial = &serial0;
     };
};
```

Chosen nodes work the same way. For example:

```
/ {
     chosen {
             zephyr,console = &serial0;
     };
};
```

To delete a property (in addition to deleting properties in general, this is how to set a boolean property to false if it's true in BOARD.dts):

```
&serial0 {
     /delete-property/ some-unwanted-property;
};
```

You can add subnodes using overlays. For example, to configure a SPI or I2C child device on an existing bus node, do something like this:

```
/* SPI device example */
&spi1 {
     my_spi_device: temp-sensor@0 {
             compatible = "...";
             label = "TEMP_SENSOR_0";
             /* reg is the chip select number, if needed;
              * If present, it must match the node's unit address. */
             reg = <0>;

             /* Configure other SPI device properties as needed.
              * Find your device's DT binding for details. */
             spi-max-frequency = <4000000>;
     };
};

/* I2C device example */
&i2c2 {
     my_i2c_device: touchscreen@76 {
             compatible = "...";
             label = "TOUCHSCREEN";
             /* reg is the I2C device address.
              * It must match the node's unit address. */
             reg = <76>;

             /* Configure other I2C device properties as needed.
              * Find your device's DT binding for details. */
     };
};
```

Other bus devices can be configured similarly:

- create the device as a subnode of the parent bus
- set its properties according to its binding

Assuming you have a suitable device driver associated with the `my_spi_device` and `my_i2c_device` compatibles, you should now be able to enable the driver via Kconfig and *get the struct device* for your newly added bus node, then use it with that driver API.

Write device drivers using devicetree APIs

"Devicetree-aware" *device drivers* should create a `struct device` for each `status = "okay"` devicetree node with a particular *compatible* (or related set of compatibles) supported by the driver.

Note: Historically, Zephyr has used Kconfig options like `CONFIG_I2C_0` and `CONFIG_I2C_1` to enable driver support for individual devices of some type. For example, if `CONFIG_I2C_1=y`, the SoC's I2C peripheral driver would create a `struct device` for "I2C bus controller number 1".

This style predates support for devicetree in Zephyr and its use is now discouraged. Existing device drivers may be made "devicetree-aware" in future releases.

Writing a devicetree-aware driver begins by defining a *devicetree binding* for the devices supported by the driver. Use existing bindings from similar drivers as a starting point. A skeletal binding to get started needs nothing more than this:

```
description: <Human-readable description of your binding>
compatible: "foo-company,bar-device"
include: base.yaml
```

See *Find a devicetree binding* for more advice on locating existing bindings.

After writing your binding, your driver C file can then use the devicetree API to find `status = "okay"` nodes with the desired compatible, and instantiate a `struct device` for each one. There are two options for instantiating each `struct device`: using instance numbers, and using node labels.

In either case:

- Each `struct device`'s name should be set to its devicetree node's `label` property. This allows the driver's users to *Get a struct device from a devicetree node* in the usual way.
- Each device's initial configuration should use values from devicetree properties whenever practical. This allows users to configure the driver using *devicetree overlays*.

Examples for how to do this follow. They assume you've already implemented the device-specific configuration and data structures and API functions, like this:

```
/* my_driver.c */
#include <drivers/some_api.h>

/* Define data (RAM) and configuration (ROM) structures: */
struct my_dev_data {
     /* per-device values to store in RAM */
};
struct my_dev_cfg {
     uint32_t freq; /* Just an example: initial clock frequency in Hz */
     /* other configuration to store in ROM */
};

/* Implement driver API functions (drivers/some_api.h callbacks): */
static int my_driver_api_func1(const struct device *dev, uint32_t *foo) { /* ... */ }
static int my_driver_api_func2(const struct device *dev, uint64_t bar) { /* ... */ }
static struct some_api my_api_funcs = {
     .func1 = my_driver_api_func1,
     .func2 = my_driver_api_func2,
};
```

Option 1: create devices using instance numbers Use this option, which uses *Instance-based APIs*, if possible. However, they only work when devicetree nodes for your driver's `compatible` are all equivalent, and you do not need to be able to distinguish between them.

To use instance-based APIs, begin by defining `DT_DRV_COMPAT` to the lowercase-and-underscores version of the compatible that the device driver supports. For example, if your driver's compatible is `"vnd,my-device"` in devicetree, you would define `DT_DRV_COMPAT` to `vnd_my_device` in your driver C file:

```
/*
 * Put this near the top of the file. After the includes is a good place.
 * (Note that you can therefore run "git grep DT_DRV_COMPAT drivers" in
 * the zephyr Git repository to look for example drivers using this style).
 */
#define DT_DRV_COMPAT vnd_my_device
```

Important: As shown, the DT_DRV_COMPAT macro should have neither quotes nor special characters. Remove quotes and convert special characters to underscores when creating `DT_DRV_COMPAT` from the compatible property.

Finally, define an instantiation macro, which creates each `struct device` using instance numbers. Do this after defining `my_api_funcs`.

```
/*
 * This instantiation macro is named "CREATE_MY_DEVICE".
 * Its "inst" argument is an arbitrary instance number.
 *
 * Put this near the end of the file, e.g. after defining "my_api_funcs".
 */
#define CREATE_MY_DEVICE(inst)                                          \
    static struct my_dev_data my_data_##inst = {                        \
            /* initialize RAM values as needed, e.g.: */                \
            .freq = DT_INST_PROP(inst, clock_frequency),                \
    };                                                                  \
    static const struct my_dev_cfg my_cfg_##inst = {                    \
            /* initialize ROM values as needed. */                      \
    };                                                                  \
    DEVICE_DT_INST_DEFINE(inst,                                         \
                          my_dev_init_function,                         \
                          NULL,                                         \
                          &my_data_##inst,                              \
                          &my_cfg_##inst,                               \
                          MY_DEV_INIT_LEVEL, MY_DEV_INIT_PRIORITY,      \
                          &my_api_funcs);
```

Notice the use of APIs like *DT_INST_PROP()* and *DEVICE_DT_INST_DEFINE()* to access devicetree node data. These APIs retrieve data from the devicetree for instance number `inst` of the node with compatible determined by `DT_DRV_COMPAT`.

Finally, pass the instantiation macro to *DT_INST_FOREACH_STATUS_OKAY()*:

```
/* Call the device creation macro for each instance: */
DT_INST_FOREACH_STATUS_OKAY(CREATE_MY_DEVICE)
```

`DT_INST_FOREACH_STATUS_OKAY` expands to code which calls `CREATE_MY_DEVICE` once for each enabled node with the compatible determined by `DT_DRV_COMPAT`. It does not append a semicolon to the end of the expansion of `CREATE_MY_DEVICE`, so the macro's expansion must end in a semicolon or function definition to support multiple devices.

Option 2: create devices using node labels Some device drivers cannot use instance numbers. One example is an SoC peripheral driver which relies on vendor HAL APIs specialized for individual IP blocks to implement Zephyr driver callbacks. Cases like this should use *DT_NODELABEL()* to refer to individual nodes in the devicetree representing the supported peripherals on the SoC. The devicetree.h *Generic APIs* can then be used to access node data.

For this to work, your *SoC's dtsi file* must define node labels like `mydevice0`, `mydevice1`, etc. appropriately for the IP blocks your driver supports. The resulting devicetree usually looks something like this:

```
/ {
        soc {
                mydevice0: dev@0 {
                        compatible = "vnd,my-device";
                };
                mydevice1: dev@1 {
                        compatible = "vnd,my-device";
                };
        };
};
```

The driver can use the `mydevice0` and `mydevice1` node labels in the devicetree to operate on specific device nodes:

```
/*
 * This is a convenience macro for creating a node identifier for
 * the relevant devices. An example use is MYDEV(0) to refer to
 * the node with label "mydevice0".
 */
#define MYDEV(idx) DT_NODELABEL(mydevice ## idx)

/*
 * Define your instantiation macro; "idx" is a number like 0 for mydevice0
 * or 1 for mydevice1. It uses MYDEV() to create the node label from the
 * index.
 */
#define CREATE_MY_DEVICE(idx)                                          \
     static struct my_dev_data my_data_##idx = {                       \
             /* initialize RAM values as needed, e.g.: */              \
             .freq = DT_PROP(MYDEV(idx), clock_frequency),             \
     };                                                                \
     static const struct my_dev_cfg my_cfg_##idx = { /* ... */ };      \
     DEVICE_DT_DEFINE(MYDEV(idx),                                      \
                      my_dev_init_function,                            \
                      NULL,                                            \
                      &my_data_##idx,                                  \
                      &my_cfg_##idx,                                   \
                      MY_DEV_INIT_LEVEL, MY_DEV_INIT_PRIORITY,         \
                      &my_api_funcs)
```

Notice the use of APIs like *DT_PROP()* and *DEVICE_DT_DEFINE()* to access devicetree node data.

Finally, manually detect each enabled devicetree node and use `CREATE_MY_DEVICE` to instantiate each `struct device`:

```
#if DT_NODE_HAS_STATUS(DT_NODELABEL(mydevice0), okay)
CREATE_MY_DEVICE(0)
#endif

#if DT_NODE_HAS_STATUS(DT_NODELABEL(mydevice1), okay)
CREATE_MY_DEVICE(1)
#endif
```

Since this style does not use `DT_INST_FOREACH_STATUS_OKAY()`, the driver author is responsible for calling `CREATE_MY_DEVICE()` for every possible node, e.g. using knowledge about the peripherals available on supported SoCs.

Device drivers that depend on other devices

At times, one `struct device` depends on another `struct device` and requires a pointer to it. For example, a sensor device might need a pointer to its SPI bus controller device. Some advice:

- Write your devicetree binding in a way that permits use of *Hardware specific APIs* from devicetree.h if possible.
- In particular, for bus devices, your driver's binding should include a file like dts/bindings/spi/spi-device.yaml which provides common definitions for devices addressable via a specific bus. This enables use of APIs like *`DT_BUS()`* to obtain a node identifier for the bus node. You can then *Get a struct device from a devicetree node* for the bus in the usual way.

Search existing bindings and device drivers for examples.

Applications that depend on board-specific devices

One way to allow application code to run unmodified on multiple boards is by supporting a devicetree alias to specify the hardware specific portions, as is done in the blinky-sample. The application can then be configured in *BOARD.dts* files or via *devicetree overlays*.

8.11.6 Troubleshooting devicetree

Here are some tips for fixing misbehaving devicetree related code.

See *Devicetree HOWTOs* for other "HOWTO" style information.

Try again with a pristine build directory

Important: Try this first, before doing anything else.

See *Pristine Builds* for examples, or just delete the build directory completely and retry.

This is general advice which is especially applicable to debugging devicetree issues, because the outputs are created during the CMake configuration phase, and are not always regenerated when one of their inputs changes.

Make sure <devicetree.h> is included

Unlike Kconfig symbols, the `devicetree.h` header must be included explicitly.

Many Zephyr header files rely on information from devicetree, so including some other API may transitively include `devicetree.h`, but that's not guaranteed.

Make sure you're using the right names

Remember that:

- In C/C++, devicetree names must be lowercased and special characters must be converted to underscores. Zephyr's generated devicetree header has DTS names converted in this way into the C tokens used by the preprocessor-based `<devicetree.h>` API.
- In overlays, use devicetree node and property names the same way they would appear in any DTS file. Zephyr overlays are just DTS fragments.

For example, if you're trying to **get** the `clock-frequency` property of a node with path `/soc/i2c@12340000` in a C/C++ file:

```
/*
 * foo.c: lowercase-and-underscores names
 */

/* Don't do this: */
#define MY_CLOCK_FREQ DT_PROP(DT_PATH(soc, i2c@1234000), clock-frequency)
/*                                          ^               ^
 *                                    @ should be _     - should be _  */

/* Do this instead: */
#define MY_CLOCK_FREQ DT_PROP(DT_PATH(soc, i2c_1234000), clock_frequency)
/*                                          ^               ^           */
```

And if you're trying to **set** that property in a devicetree overlay:

```
/*
 * foo.overlay: DTS names with special characters, etc.
 */

/* Don't do this; you'll get devicetree errors. */
&{/soc/i2c_12340000/} {
     clock_frequency = <115200>;
};

/* Do this instead. Overlays are just DTS fragments. */
&{/soc/i2c@12340000/} {
     clock-frequency = <115200>;
};
```

Look at the preprocessor output

To save preprocessor output when using GCC-based toolchains, add `-save-temps=obj` to the `EXTRA_CFLAGS` CMake variable. For example, to build hello_world with west with this option set, use:

```
west build -b BOARD samples/hello_world -- -DEXTRA_CFLAGS=-save-temps=obj
```

This will create a preprocessor output file named `foo.c.i` in the build directory for each source file `foo.c`.

You can then search for the file in the build directory to see what your devicetree macros expanded to. For example, on macOS and Linux, using `find` to find `main.c.i`:

```
$ find build -name main.c.i
build/CMakeFiles/app.dir/src/main.c.i
```

It's usually easiest to run a style formatter on the results before opening them. For example, to use `clang-format` to reformat the file in place:

```
clang-format -i build/CMakeFiles/app.dir/src/main.c.i
```

You can then open the file in your favorite editor to view the final C results after preprocessing.

Validate properties

If you're getting a compile error reading a node property, check your node identifier and property. For example, if you get a build error on a line that looks like this:

```
int baud_rate = DT_PROP(DT_NODELABEL(my_serial), current_speed);
```

Try checking the node by adding this to the file and recompiling:

```
#if !DT_NODE_EXISTS(DT_NODELABEL(my_serial))
#error "whoops"
#endif
```

If you see the "whoops" error message when you rebuild, the node identifier isn't referring to a valid node. *Get your devicetree and generated header* and debug from there.

Some hints for what to check next if you don't see the "whoops" error message:

- did you *Make sure you're using the right names*?
- does the *property exist*?
- does the node have a *matching binding*?
- does the binding define the property?

Check for missing bindings

See *Devicetree bindings* for information about bindings, and *Bindings index* for information on bindings built into Zephyr.

If the build fails to *Find a devicetree binding* for a node, then either the node's `compatible` property is not defined, or its value has no matching binding. If the property is set, check for typos in its name. In a devicetree source file, `compatible` should look like `"vnd,some-device"` – *Make sure you're using the right names*.

If your binding file is not under `zephyr/dts`, you may need to set *DTS_ROOT*; see *Where bindings are located*.

Errors with DT_INST_() APIs

If you're using an API like *DT_INST_PROP()*, you must define `DT_DRV_COMPAT` to the lowercase-and-underscores version of the compatible you are interested in. See *Option 1: create devices using instance numbers*.

8.11.7 Devicetree versus Kconfig

Along with devicetree, Zephyr also uses the Kconfig language to configure the source code. Whether to use devicetree or Kconfig for a particular purpose can sometimes be confusing. This section should help you decide which one to use.

In short:

- Use devicetree to describe **hardware** and its **boot-time configuration**. Examples include peripherals on a board, boot-time clock frequencies, interrupt lines, etc.
- Use Kconfig to configure **software support** to build into the final image. Examples include whether to add networking support, which drivers are needed by the application, etc.

In other words, devicetree mainly deals with hardware, and Kconfig with software.

For example, consider a board containing a SoC with 2 UART, or serial port, instances.

- The fact that the board has this UART **hardware** is described with two UART nodes in the devicetree. These provide the UART type (via the `compatible` property) and certain settings such as the address range of the hardware peripheral registers in memory (via the `reg` property).
- Additionally, the UART **boot-time configuration** is also described with devicetree. This could include configuration such as the RX IRQ line's priority and the UART baud rate. These may be modifiable at runtime, but their boot-time configuration is described in devicetree.
- Whether or not to include **software support** for UART in the build is controlled via Kconfig. Applications which do not need to use the UARTs can remove the driver source code from the build using Kconfig, even though the board's devicetree still includes UART nodes.

As another example, consider a device with a 2.4GHz, multi-protocol radio supporting both the Bluetooth Low Energy and 802.15.4 wireless technologies.

- Devicetree should be used to describe the presence of the radio **hardware**, what driver or drivers it's compatible with, etc.
- **Boot-time configuration** for the radio, such as TX power in dBm, should also be specified using devicetree.
- Kconfig should determine which **software features** should be built for the radio, such as selecting a BLE or 802.15.4 protocol stack.

As another example, Kconfig options that formerly enabled a particular instance of a driver (that is itself enabled by Kconfig) have been removed. The devices are selected individually using devicetree's *status* keyword on the corresponding hardware instance.

There are **exceptions** to these rules:

- Because Kconfig is unable to flexibly control some instance-specific driver configuration parameters, such as the size of an internal buffer, these options may be defined in devicetree. However, to make clear that they are specific to Zephyr drivers and not hardware description or configuration these properties should be prefixed with `zephyr,`, e.g. `zephyr,random-mac-address` in the common Ethernet devicetree properties.
- Devicetree's `chosen` keyword, which allows the user to select a specific instance of a hardware device to be used for a particular purpose. An example of this is selecting a particular UART for use as the system's console.

8.12 Peripheral and Hardware Emulators

8.12.1 Overview

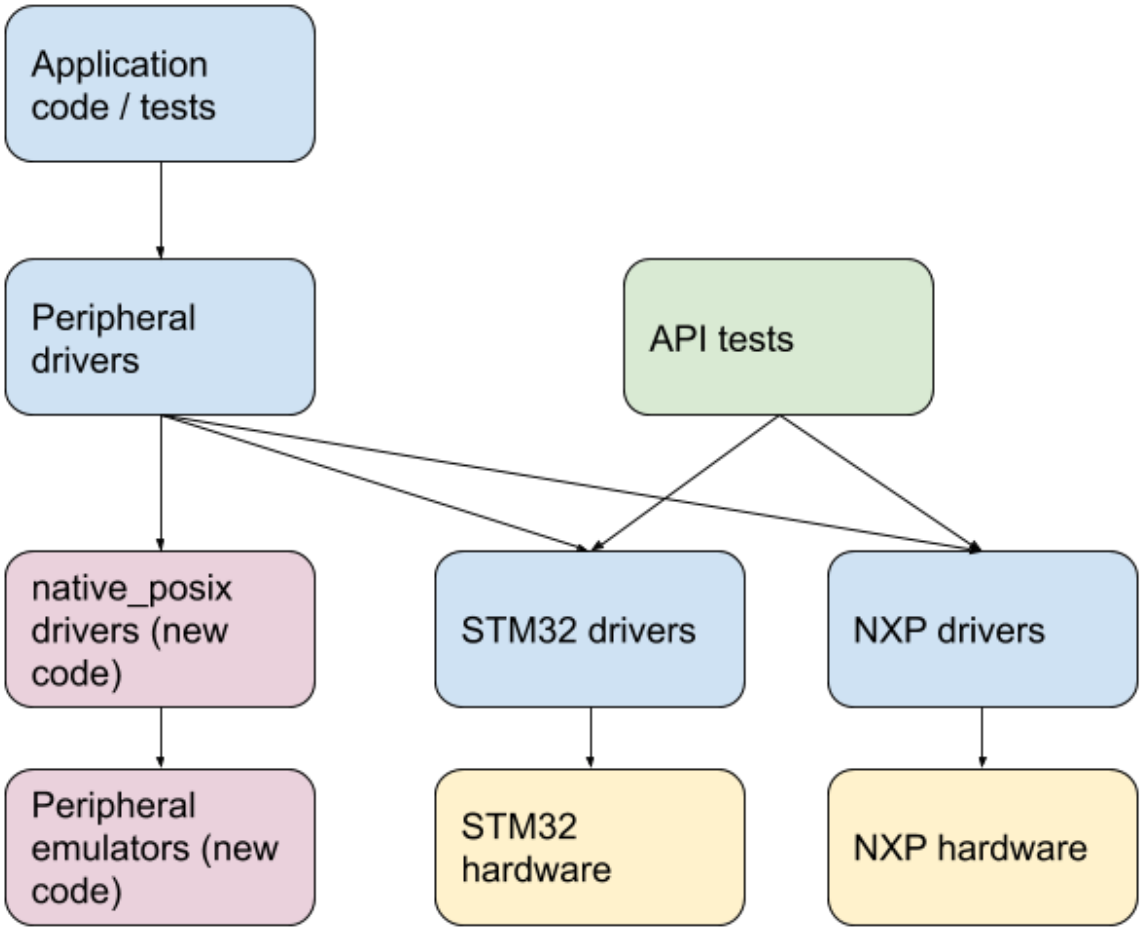
Zephyr supports a simple emulator framework to support testing of drivers without requiring real hardware.

Emulators are used to emulate hardware devices, to support testing of various subsystems. For example, it is possible to write an emulator for an I2C compass such that it appears on the I2C bus and can be used just like a real hardware device.

Emulators often implement special features for testing. For example a compass may support returning bogus data if the I2C bus speed is too high, or may return invalid measurements if calibration has not yet been completed. This allows for testing that high-level code can handle these situations correctly. Test coverage can therefore approach 100% if all failure conditions are emulated.

8.12.2 Concept

The diagram below shows application code / high-level tests at the top. This is the ultimate application we want to run.

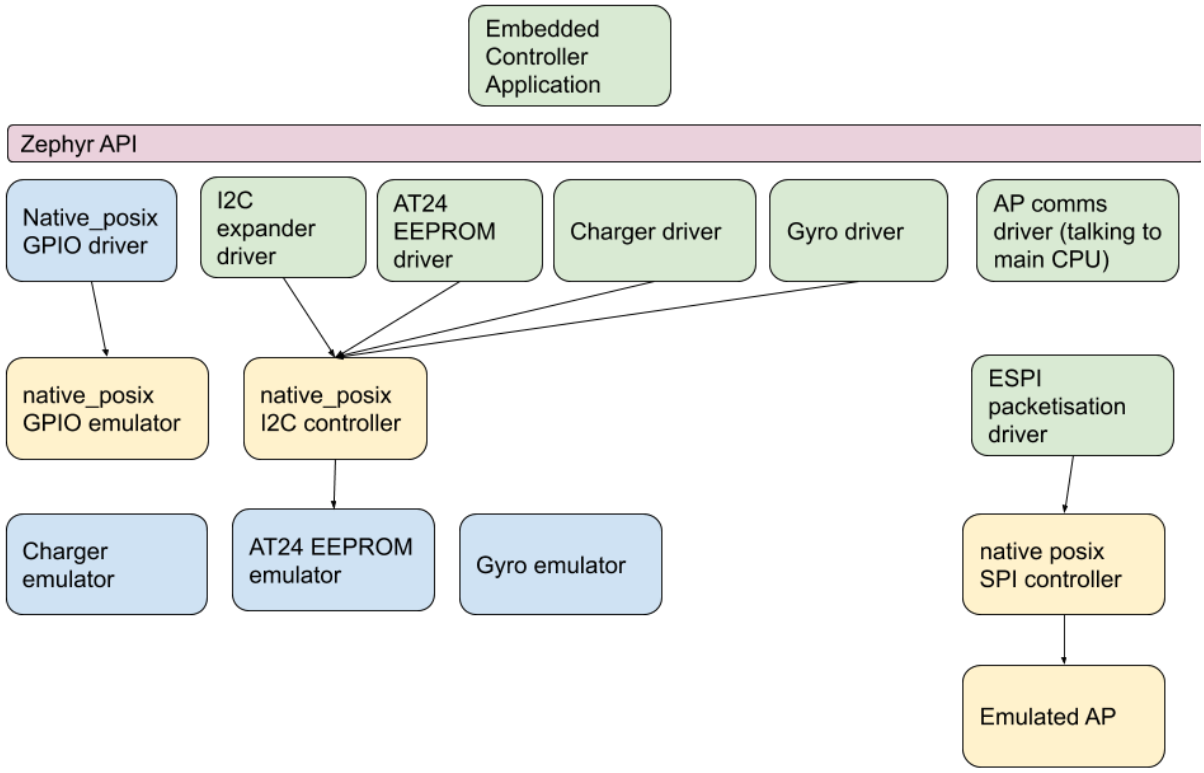

Application code / tests
Peripheral drivers
API tests
native_posix drivers (new code)
STM32 drivers
NXP drivers
Peripheral emulators (new code)
STM32 hardware
NXP hardware

Below that are peripheral drivers, such as the AT24 EEPROM driver. We can test peripheral drivers using an emulation driver connected via a native_posix I2C controller/emulator which passes I2C traffic from the AT24 driver to the AT24 simulator.

Separately we can test the STM32 and NXP I2C drivers on real hardware using API tests. These require some sort of device attached to the bus, but with this, we can validate much of the driver functionality.

Putting the two together, we can test the application and peripheral code entirely on native_posix. Since we know that the I2C driver on the real hardware works, we should expect the application and peripheral drivers to work on the real hardware also.

Using the above framework we can test an entire application (e.g. Embedded Controller) on native_posix using emulators for all non-chip drivers:



The 'real' code is shown in green. The Zephyr emulation-framework code is shown in yellow. The blue boxes are the extra code we have to write to emulate the peripherals.

With this approach we can:

- Write individual tests for each driver (green), covering all failure modes, error conditions, etc.
- Ensure 100% test coverage for drivers (green)
- Write tests for combinations of drivers, such as GPIOs provided by an I2C GPIO expander driver talking over an I2C bus, with the GPIOs controlling a charger. All of this can work in the emulated environment or on real hardware.
- Write a complex application that ties together all of these pieces and runs on native_posix. We can develop on a host, use source-level debugging, etc.
- Transfer the application to any board which provides the required features (e.g. I2C, enough GPIOs), by adding Kconfig and devicetree fragments.

8.12.3 Available emulators

Zephyr includes the following emulators:

- EEPROM, which uses a file as the EEPROM contents
- I2C emulator driver, allowing drivers to be connected to an emulator so that tests can be performed without access to the real hardware
- SPI emulator driver, which does the same for SPI
- eSPI emulator driver, which does the same for eSPI. The emulator is being developed to support more functionalities.

A GPIO emulator is planned but is not yet complete.

8.12.4 Samples

Here are some examples present in Zephyr:

1. Bosche BMI160 sensor driver connected via both I2C and SPI to an emulator:

```
west build -b native_posix tests/drivers/sensor/accel/
```

2. Simple test of the EEPROM emulator:

```
west build -b native_posix tests/drivers/eeprom
```

3. The same test has a second EEPROM which is an Atmel AT24 EEPROM driver connected via I2C an emulator:

```
west build -b native_posix tests/drivers/eeprom
```

8.13 Modules (External projects)

Zephyr relies on the source code of several externally maintained projects in order to avoid reinventing the wheel and to reuse as much well-established, mature code as possible when it makes sense. In the context of Zephyr's build system those are called *modules*. These modules must be integrated with the Zephyr build system, as described in more detail in other sections on this page.

To be classified as a candidate for being included in the default list of modules, an external project is required to have its own life-cycle outside the Zephyr Project, that is, reside in its own repository, and have its own contribution and maintenance workfow and release process. Zephyr modules should not contain code that is written exclusively for Zephyr. Instead, such code should be contributed to the main zephyr tree.

Modules to be included in the default manifest of the Zephyr project need to provide functionality or features endorsed and approved by the project Technical Steering Committee and should comply with the *module licensing requirements* and *contribution guidelines*. They should also have a Zephyr developer that is committed to maintain the module codebase.

Zephyr depends on several categories of modules, including but not limited to:

- Debugger integration
- Silicon vendor Hardware Abstraction Layers (HALs)
- Cryptography libraries
- File Systems
- Inter-Process Communication (IPC) libraries

This page summarizes a list of policies and best practices which aim at better organizing the workflow in Zephyr modules.

8.13.1 Module Repositories

- All modules included in the default manifest shall be hosted in repositories under the zephyrproject-rtos GitHub organization.
- The module repository codebase shall include a *module.yml* file in a `zephyr/` folder at the root of the repository.
- Module repository names should follow the convention of using lowercase letters and dashes instead of underscores. This rule will apply to all new module repositories, except for repositories that are directly tracking external projects (hosted in Git repositories); such modules may be named as their external project counterparts.

  Note: Existing module repositories that do not conform to the above convention do not need to be renamed to comply with the above convention.

- Modules should use "zephyr" as the default name for the repository main branch. Branches for specific purposes, for example, a module branch for an LTS Zephyr version, shall have names starting with the 'zephyr_' prefix.
- If the module has an external (upstream) project repository, the module repository should preserve the upstream repository folder structure.

  Note: It is not required in module repositories to maintain a 'master' branch mirroring the master branch of the external repository. It is not recommended as this may generate confusion around the module's main branch, which should be 'zephyr'.

Synchronizing with upstream

It is preferred to synchronize a module respository with the latest stable release of the corresponding external project. It is permitted, however, to update a Zephyr module repository with the latest development branch tip, if this is required to get important updates in the module codebase. When synchronizing a module with upstream it is mandatory to document the rationale for performing the particular update.

Requirements for allowed practices Changes to the main branch of a module repository, including synchronization with upstream code base, may only be applied via pull requests. These pull requests shall be *verifiable* by Zephyr CI and *mergeable* (e.g. with the *Rebase and merge*, or *Create a merge commit* option using Github UI). This ensures that the incoming changes are always **reviewable**, and the *downstream* module repository history is incremental (that is, existing commits, tags, etc. are always preserved). This policy also allows to run Zephyr CI, git lint, identity, and license checks directly on the set of changes that are to be brought into the module repository.

Note: Force-pushing to a module's main branch is not allowed.

Allowed practices The following practices conform to the above requirements and should be followed in all modules repositories. It is up to the module code owner to select the preferred synchronization practice, however, it is required that the selected practice is consistently followed in the respective module repository.

Updating modules with a diff from upstream: Upstream changes brought as a single *snapshot* commit (manual diff) in a pull request against the module's main branch, which may be merged using the *Rebase & merge* operation. This approach is simple and should be applicable to all modules with the downside of supressing the upstream history in the module repository.

> **Note:** The above practice is the only allowed practice in modules where the external project is not hosted in an upstream Git repository.

The commit message is expected to identify the upstream project URL, the version to which the module is updated (upstream version, tag, commit SHA, if applicable, etc.), and the reason for the doing the update.

Updating modules by merging the upstream branch: Upstream changes brought in by performing a Git merge of the intended upstream branch (e.g. main branch, latest release branch, etc.) submitting the result in pull request against the module main branch, and merging the pull request using the *Create a merge commit* operation. This approach is applicable to modules with an upstream project Git repository. The main advantages of this approach is that the upstream repository history (that is, the original commit SHAs) is preserved in the module repository. The downside of this approach is that two additional merge commits are generated in the downstream main branch.

8.13.2 Contributing to Zephyr modules

Individual Roles & Responsibilities

To facilitate management of Zephyr module repositories, the following individual roles are defined.

Administrator: Each Zephyr module shall have an administrator who is responsible for managing access to the module repository, for example, for adding individuals as Collaborators in the repository at the request of the module owner. Module administrators are members of the Administrators team, that is a group of project members with admin rights to module GitHub repositories.

Module owner: Each module shall have a module code owner. Module owners will have the overall responsibility of the contents of a Zephyr module repository. In particular, a module owner will:

- coordinate code reviewing in the module repository
- be the default assignee in pull-requests against the repository's main branch
- request additional collaborators to be added to the repository, as they see fit
- regularly synchronize the module repository with its upstream counterpart following the policies described in *Synchronizing with upstream*
- be aware of security vulnerability issues in the external project and update the module repository to include security fixes, as soon as the fixes are available in the upstream code base
- list any known security vulnerability issues, present in the module codebase, in Zephyr release notes.

> **Note:** Module owners are not required to be Zephyr *Maintainers*.

Merger: The Zephyr Release Engineering team has the right and the responsibility to merge approved pull requests in the main branch of a module repository.

Maintaining the module codebase

Updates in the zephyr main tree, for example, in public Zephyr APIs, may require patching a module's codebase. The responsibility for keeping the module codebase up to date is shared between the **contributor** of such updates in Zephyr and the module **owner**. In particular:

- the contributor of the original changes in Zephyr is required to submit the corresponding changes that are required in module repositories, to ensure that Zephyr CI on the pull request with the original changes, as well as the module integration testing are successful.

- the module owner has the overall responsibility for synchronizing and testing the module codebase with the zephyr main tree. This includes occasional advanced testing of the module's codebase in addition to the testing performed by Zephyr's CI. The module owner is required to fix issues in the module's codebase that have not been caught by Zephyr pull request CI runs.

Contributing changes to modules

Submitting and merging changes directly to a module's codebase, that is, before they have been merged in the corresponding external project repository, should be limited to:

- changes required due to updates in the zephyr main tree
- urgent changes that should not wait to be merged in the external project first, such as fixes to security vulnerabilities.

Non-trivial changes to a module's codebase, including changes in the module design or functionality should be discouraged, if the module has an upstream project repository. In that case, such changes shall be submitted to the upstream project, directly.

Submitting changes to modules describes in detail the process of contributing changes to module repositories.

Contribution guidelines Contributing to Zephyr modules shall follow the generic project *Contribution guidelines*.

Pull Requests: may be merged with minimum of 2 approvals, including an approval by the PR assignee. In addition to this, pull requests in module repositories may only be merged if the introduced changes are verified with Zephyr CI tools, as described in more detail in other sections on this page.

The merging of pull requests in the main branch of a module repository must be coupled with the corresponding manifest file update in the zephyr main tree.

Issue Reporting: GitHub issues are intentionally disabled in module repositories, in favor of a centralized policy for issue reporting. Tickets concerning, for example, bugs or enhancements in modules shall be opened in the main zephyr repository. Issues should be appropriately labeled using GitHub labels corresponding to each module, where applicable.

Note: It is allowed to file bug reports for zephyr modules to track the corresponding upstream project bugs in Zephyr. These bug reports shall not affect the *Release Quality Criteria*.

8.13.3 Licensing requirements and policies

All source files in a module's codebase shall include a license header, unless the module repository has **main license file** that covers source files that do not include license headers.

Main license files shall be added in the module's codebase by Zephyr developers, only if they exist as part of the external project, and they contain a permissive OSI-compliant license. Main license files should preferably contain the full license text instead of including an SPDX license identifier. If multiple main license files are present it shall be made clear which license applies to each source file in a module's codebase.

Individual license headers in module source files supersede the main license.

Any new content to be added in a module repository will require to have license coverage.

Note: Zephyr recommends conveying module licensing via individual license headers and main license files. This not a hard requirement; should an external project have its own practice of conveying how licensing applies in the module's codebase (for example, by having

a single or multiple main license files), this practice may be accepted by and be referred to in the Zephyr module, as long as licensing requirements, for example OSI compliance, are satisfied.

License policies

When creating a module repository a developer shall:

- import the main license files, if they exist in the external project, and
- document (for example in the module README or .yml file) the default license that covers the module's codebase.

License checks License checks (via CI tools) shall be enabled on every pull request that adds new content in module repositories.

8.13.4 Documentation requirements

All Zephyr module repositories shall include an .rst file documenting:

- the scope and the purpose of the module
- how the module integrates with Zephyr
- the owner of the module repository
- synchronization information with the external project (commit, SHA, version etc.)
- licensing information as described in *Licensing requirements and policies*.

The file shall be required for the inclusion of the module and the contained information should be kept up to date.

8.13.5 Testing requirements

All Zephyr modules should provide some level of **integration** testing, ensuring that the integration with Zephyr works correctly. Integration tests:

- may be in the form of a minimal set of samples and tests that reside in the zephyr main tree
- should verify basic usage of the module (configuration, functional APIs, etc.) that is integrated with Zephyr.
- shall be built and executed (for example in QEMU) as part of twister runs in pull requests that introduce changes in module repositories.

Note: New modules, that are candidates for being included in the Zephyr default manifest, shall provide some level of integration testing.

Note: Vendor HALs are implicitly tested via Zephyr tests built or executed on target platforms, so they do not need to provide integration tests.

The purpose of integration testing is not to provide functional verification of the module; this should be part of the testing framework of the external project.

Certain external projects provide test suites that reside in the upstream testing infrastructure but are written explicitly for Zephyr. These tests may (but are not required to) be part of the Zephyr test framework.

8.13.6 Deprecating and removing modules

Modules may be deprecated for reasons including, but not limited to:

- Lack of maintainership in the module
- Licensing changes in the external project
- Codebase becoming obsolete

The module information shall indicate whether a module is deprecated and the build system shall issue a warning when trying to build Zephyr using a deprecated module.

Deprecated modules may be removed from the Zephyr default manifest after 2 Zephyr releases.

> **Note:** Repositories of removed modules shall remain accessible via their original URL, as they are required by older Zephyr versions.

8.13.7 Integrate modules in Zephyr build system

The build system variable `ZEPHYR_MODULES` is a CMake list of absolute paths to the directories containing Zephyr modules. These modules contain `CMakeLists.txt` and `Kconfig` files describing how to build and configure them, respectively. Module `CMakeLists.txt` files are added to the build using CMake's add_subdirectory() command, and the `Kconfig` files are included in the build's Kconfig menu tree.

If you have *west* installed, you don't need to worry about how this variable is defined unless you are adding a new module. The build system knows how to use west to set `ZEPHYR_MODULES`. You can add additional modules to this list by setting the `ZEPHYR_EXTRA_MODULES` CMake variable or by adding a `ZEPHYR_EXTRA_MODULES` line to `.zephyrrc` (See the section on *Setting Variables* for more details). This can be useful if you want to keep the list of modules found with west and also add your own.

> **Note:** If the module `FOO` is provided by *west* but also given with `-DZEPHYR_EXTRA_MODULES=/<path>/foo` then the module given by the command line variable `ZEPHYR_EXTRA_MODULES` will take precedence. This allows you to use a custom version of `FOO` when building and still use other Zephyr modules provided by *west*. This can for example be useful for special test purposes.

See *Basics* for more on west workspaces.

Finally, you can also specify the list of modules yourself in various ways, or not use modules at all if your application doesn't need them.

8.13.8 Module yaml file description

A module can be described using a file named `zephyr/module.yml`. The format of `zephyr/module.yml` is described in the following:

Module name

Each Zephyr module is given a name by which it can be referred to in the build system.

The name may be specified in the `zephyr/module.yml` file:

```
name: <name>
```

In CMake the location of the Zephyr module can then be referred to using the CMake variable ZEPHYR_<MODULE_NAME>_MODULE_DIR and the variable ZEPHYR_<MODULE_NAME>_CMAKE_DIR holds the location of the directory containing the module's CMakeLists.txt file.

Note: When used for CMake and Kconfig variables, all letters in module names are converted to uppercase and all non-alphanumeric characters are converted to underscores (_). As example, the module foo-bar must be referred to as ZEPHYR_FOO_BAR_MODULE_DIR in CMake and Kconfig.

Here is an example for the Zephyr module foo:

```
name: foo
```

Note: If the name field is not specified then the Zephyr module name will be set to the name of the module folder. As example, the Zephyr module located in <workspace>/modules/bar will use bar as its module name if nothing is specified in zephyr/module.yml.

Module integration files (in-module)

Inclusion of build files, CMakeLists.txt and Kconfig, can be described as:

```
build:
  cmake: <cmake-directory>
  kconfig: <directory>/Kconfig
```

The cmake: <cmake-directory> part specifies that <cmake-directory> contains the CMakeLists.txt to use. The kconfig: <directory>/Kconfig part specifies the Kconfig file to use. Neither is required: cmake defaults to zephyr, and kconfig defaults to zephyr/Kconfig.

Here is an example module.yml file referring to CMakeLists.txt and Kconfig files in the root directory of the module:

```
build:
  cmake: .
  kconfig: Kconfig
```

Build system integration

When a module has a module.yml file, it will automatically be included into the Zephyr build system. The path to the module is then accessible through Kconfig and CMake variables.

In both Kconfig and CMake, the variable ZEPHYR_<MODULE_NAME>_MODULE_DIR contains the absolute path to the module.

In CMake, ZEPHYR_<MODULE_NAME>_CMAKE_DIR contains the absolute path to the directory containing the CMakeLists.txt file that is included into CMake build system. This variable's value is empty if the module.yml file does not specify a CMakeLists.txt.

To read these variables for a Zephyr module named foo:

- In CMake: use ${ZEPHYR_FOO_MODULE_DIR} for the module's top level directory, and ${ZEPHYR_FOO_CMAKE_DIR} for the directory containing its CMakeLists.txt
- In Kconfig: use $(ZEPHYR_FOO_MODULE_DIR) for the module's top level directory

Notice how a lowercase module name `foo` is capitalized to `FOO` in both CMake and Kconfig.

These variables can also be used to test whether a given module exists. For example, to verify that `foo` is the name of a Zephyr module:

```
if(ZEPHYR_FOO_MODULE_DIR)
  # Do something if FOO exists.
endif()
```

In Kconfig, the variable may be used to find additional files to include. For example, to include the file `some/Kconfig` in module `foo`:

```
source "$(ZEPHYR_FOO_MODULE_DIR)/some/Kconfig"
```

During CMake processing of each Zephyr module, the following two variables are also available:

- the current module's top level directory: `${ZEPHYR_CURRENT_MODULE_DIR}`
- the current module's `CMakeLists.txt` directory: `${ZEPHYR_CURRENT_CMAKE_DIR}`

This removes the need for a Zephyr module to know its own name during CMake processing. The module can source additional CMake files using these `CURRENT` variables. For example:

```
include(${ZEPHYR_CURRENT_MODULE_DIR}/cmake/code.cmake)
```

It is possible to append values to a Zephyr CMake list variable from the module's first CMakeLists.txt file. To do so, append the value to the list and then set the list in the PARENT_SCOPE of the CMakeLists.txt file. For example, to append `bar` to the `FOO_LIST` variable in the Zephyr CMakeLists.txt scope:

```
list(APPEND FOO_LIST bar)
set(FOO_LIST ${FOO_LIST} PARENT_SCOPE)
```

An example of a Zephyr list where this is useful is when adding additional directories to the `SYSCALL_INCLUDE_DIRS` list.

Zephyr module dependencies

A Zephyr module may be dependent on other Zephyr modules to be present in order to function correctly. Or it might be that a given Zephyr module must be processed after another Zephyr module, due to dependencies of certain CMake targets.

Such a dependency can be described using the `depends` field.

```
build:
  depends:
    - <module>
```

Here is an example for the Zephyr module `foo` that is dependent on the Zephyr module `bar` to be present in the build system:

```
name: foo
build:
  depends:
  - bar
```

This example will ensure that `bar` is present when `foo` is included into the build system, and it will also ensure that `bar` is processed before `foo`.

Module integration files (external)

Module integration files can be located externally to the Zephyr module itself. The `MODULE_EXT_ROOT` variable holds a list of roots containing integration files located externally to Zephyr modules.

Module integration files in Zephyr The Zephyr repository contain `CMakeLists.txt` and `Kconfig` build files for certain known Zephyr modules.

Those files are located under

```
<ZEPHYR_BASE>
└── modules
    └── <module_name>
        ├── CMakeLists.txt
        └── Kconfig
```

Module integration files in a custom location You can create a similar `MODULE_EXT_ROOT` for additional modules, and make those modules known to Zephyr build system.

Create a `MODULE_EXT_ROOT` with the following structure

```
<MODULE_EXT_ROOT>
└── modules
    ├── modules.cmake
    └── <module_name>
        ├── CMakeLists.txt
        └── Kconfig
```

and then build your application by specifying `-DMODULE_EXT_ROOT` parameter to the CMake build system. The `MODULE_EXT_ROOT` accepts a CMake list of roots as argument.

A Zephyr module can automatically be added to the `MODULE_EXT_ROOT` list using the module description file `zephyr/module.yml`, see *Build settings*.

Note: `ZEPHYR_BASE` is always added as a `MODULE_EXT_ROOT` with the lowest priority. This allows you to overrule any integration files under `<ZEPHYR_BASE>/modules/<module_name>` with your own implementation your own `MODULE_EXT_ROOT`.

The `modules.cmake` file must contain the logic that specifies the integration files for Zephyr modules via specifically named CMake variables.

To include a module's CMake file, set the variable `ZEPHYR_<MODULE_NAME>_CMAKE_DIR` to the path containing the CMake file.

To include a module's Kconfig file, set the variable `ZEPHYR_<MODULE_NAME>_KCONFIG` to the path to the Kconfig file.

The following is an example on how to add support the the `FOO` module.

Create the following structure

```
<MODULE_EXT_ROOT>
└── modules
    ├── modules.cmake
    └── foo
        ├── CMakeLists.txt
        └── Kconfig
```

and inside the `modules.cmake` file, add the following content

```
set(ZEPHYR_FOO_CMAKE_DIR ${CMAKE_CURRENT_LIST_DIR}/foo)
set(ZEPHYR_FOO_KCONFIG   ${CMAKE_CURRENT_LIST_DIR}/foo/Kconfig)
```

Module integration files (zephyr/module.yml) The module description file `zephyr/module.yml` can be used to specify that the build files, `CMakeLists.txt` and `Kconfig`, are located in a *Module integration files (external)*.

Build files located in a `MODULE_EXT_ROOT` can be described as:

```
build:
  cmake-ext: True
  kconfig-ext: True
```

This allows control of the build inclusion to be described externally to the Zephyr module.

The Zephyr repository itself is always added as a Zephyr module ext root.

Build settings

It is possible to specify additional build settings that must be used when including the module into the build system.

All `root` settings are relative to the root of the module.

Build settings supported in the `module.yml` file are:

- `board_root`: Contains additional boards that are available to the build system. Additional boards must be located in a `<board_root>/boards` folder.
- `dts_root`: Contains additional dts files related to the architecture/soc families. Additional dts files must be located in a `<dts_root>/dts` folder.
- `soc_root`: Contains additional SoCs that are available to the build system. Additional SoCs must be located in a `<soc_root>/soc` folder.
- `arch_root`: Contains additional architectures that are available to the build system. Additional architectures must be located in a `<arch_root>/arch` folder.
- `module_ext_root`: Contains `CMakeLists.txt` and `Kconfig` files for Zephyr modules, see also *Module integration files (external)*.

Example of a `module.yaml` file containing additional roots, and the corresponding file system layout.

```
build:
  settings:
    board_root: .
    dts_root: .
    soc_root: .
    arch_root: .
    module_ext_root: .
```

requires the following folder structure:

```
<zephyr-module-root>
├── arch
├── boards
├── dts
├── modules
└── soc
```

Twister (Test Runner)

To execute both tests and samples available in modules, the Zephyr test runner (twister) should be pointed to the directories containing those samples and tests. This can be done by specifying the path to both samples and tests in the `zephyr/module.yml` file. Additionally, if a module defines out of tree boards, the module file can point twister to the path where those files are maintained in the module. For example:

```
build:
  cmake: .
samples:
  - samples
tests:
  - tests
boards:
  - boards
```

Module Inclusion

Using West If west is installed and `ZEPHYR_MODULES` is not already set, the build system finds all the modules in your west installation and uses those. It does this by running *west list* to get the paths of all the projects in the installation, then filters the results to just those projects which have the necessary module metadata files.

Each project in the `west list` output is tested like this:

- If the project contains a file named `zephyr/module.yml`, then the content of that file will be used to determine which files should be added to the build, as described in the previous section.
- Otherwise (i.e. if the project has no `zephyr/module.yml`), the build system looks for `zephyr/CMakeLists.txt` and `zephyr/Kconfig` files in the project. If both are present, the project is considered a module, and those files will be added to the build.
- If neither of those checks succeed, the project is not considered a module, and is not added to `ZEPHYR_MODULES`.

Without West If you don't have west installed or don't want the build system to use it to find Zephyr modules, you can set `ZEPHYR_MODULES` yourself using one of the following options. Each of the directories in the list must contain either a `zephyr/module.yml` file or the files `zephyr/CMakeLists.txt` and `Kconfig`, as described in the previous section.

1. At the CMake command line, like this:

   ```
   cmake -DZEPHYR_MODULES=<path-to-module1>[;<path-to-module2>[...]] ...
   ```

2. At the top of your application's top level `CMakeLists.txt`, like this:

   ```
   set(ZEPHYR_MODULES <path-to-module1> <path-to-module2> [...])
   find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
   ```

   If you choose this option, make sure to set the variable **before** calling `find_package(Zephyr ...)`, as shown above.

3. In a separate CMake script which is pre-loaded to populate the CMake cache, like this:

   ```
   # Put this in a file with a name like "zephyr-modules.cmake"
   set(ZEPHYR_MODULES <path-to-module1> <path-to-module2>
     CACHE STRING "pre-cached modules")
   ```

You can tell the build system to use this file by adding `-C zephyr-modules.cmake` to your CMake command line.

Not using modules If you don't have west installed and don't specify `ZEPHYR_MODULES` yourself, then no additional modules are added to the build. You will still be able to build any applications that don't require code or Kconfig options defined in an external repository.

8.13.9 Submitting changes to modules

When submitting new or making changes to existing modules the main repository Zephyr needs a reference to the changes to be able to verify the changes. In the main tree this is done using revisions. For code that is already merged and part of the tree we use the commit hash, a tag, or a branch name. For pull requests however, we require specifying the pull request number in the revision field to allow building the zephyr main tree with the changes submitted to the module.

To avoid merging changes to master with pull request information, the pull request should be marked as `DNM` (Do Not Merge) or preferably a draft pull request to make sure it is not merged by mistake and to allow for the module to be merged first and be assigned a permanent commit hash. Once the module is merged, the revision will need to be changed either by the submitter or by the maintainer to the commit hash of the module which reflects the changes.

Note that multiple and dependent changes to different modules can be submitted using exactly the same process. In this case you will change multiple entries of all modules that have a pull request against them.

Process for submitting a new module

Please follow the process in *Submission and review process* and obtain the TSC approval to integrate the external source code as a module

If the request is approved, a new repository will created by the project team and initialized with basic information that would allow submitting code to the module project following the project contribution guidelines.

If a module is maintained as a fork of another project on Github, the Zephyr module related files and changes in relation to upstream need to be maintained in a special branch named `zephyr`.

Maintainers from the Zephyr project will create the repository and initialize it. You will be added as a collaborator in the new repository. Submit the module content (code) to the new repository following the guidelines described *here*, and then add a new entry to the west.yml with the following information:

```
- name: <name of repository>
  path: <path to where the repository should be cloned>
  revision: <ref pointer to module pull request>
```

For example, to add *my_module* to the manifest:

```
- name: my_module
  path: modules/lib/my_module
  revision: pull/23/head
```

Where 23 in the example above indicated the pull request number submitted to the *my_module* repository. Once the module changes are reviewed and merged, the revision needs to be changed to the commit hash from the module repository.

Process for submitting changes to existing modules

1. Submit the changes using a pull request to an existing repository following the *contribution guidelines*.
2. Submit a pull request changing the entry referencing the module into the west.yml of the main Zephyr tree with the following information:

```
- name: <name of repository>
  path: <path to where the repository should be cloned>
  revision: <ref pointer to module pull request>
```

For example, to add *my_module* to the manifest:

```
- name: my_module
  path: modules/lib/my_module
  revision: pull/23/head
```

Where 23 in the example above indicated the pull request number submitted to the *my_module* repository. Once the module changes are reviewed and merged, the revision needs to be changed to the commit hash from the module repository.

8.14 Networking

The networking section contains information regarding the network stack of the Zephyr kernel. Use the information to understand the principles behind the operation of the stacks and how they were implemented.

8.14.1 Overview

Supported Features

The networking IP stack is modular and highly configurable via build-time configuration options. You can minimize system memory consumption by enabling only those network features required by your application. Almost all features can be disabled if not needed.

- **IPv6** The support for IPv6 is enabled by default. Various IPv6 sub-options can be enabled or disabled depending on networking needs.
  - Developer can set the number of unicast and multicast IPv6 addresses that are active at the same time.
  - The IPv6 address for the device can be set either statically or dynamically using SLAAC (Stateless Address Auto Configuration) (RFC 4862).
  - The system also supports multiple IPv6 prefixes and the maximum IPv6 prefix count can be configured at build time.
  - The IPv6 neighbor cache can be disabled if not needed, and its size can be configured at build time.
  - The IPv6 neighbor discovery support (RFC 4861) is enabled by default.

  - Multicast Listener Discovery v2 support (RFC 3810) is enabled by default.
  - IPv6 header compression (6lo) is available for IPv6 connectivity for Bluetooth IPSP (RFC 7668) and IEEE 802.15.4 networks (RFC 4944).

- **IPv4** The legacy IPv4 is supported by the networking stack. It cannot be used by IEEE 802.15.4 or Bluetooth IPSP as those network technologies support only IPv6. IPv4 can be used in Ethernet based networks. By default IPv4 support is disabled.
  - DHCP (Dynamic Host Configuration Protocol) client is supported (RFC 2131).
  - The IPv4 address can also be configured manually. Static IPv4 addresses are supported by default.
- **Dual stack support.** The networking stack allows a developer to configure the system to use both IPv6 and IPv4 at the same time.
- **UDP** User Datagram Protocol (RFC 768) is supported. The developer can send UDP datagrams (client side support) or create a listener to receive UDP packets destined to certain port (server side support).
- **TCP** Transmission Control Protocol (RFC 793) is supported. Both server and client roles can be used the the application. The amount of TCP sockets that are available to applications can be configured at build time.
- **BSD Sockets API** Support for a subset of a *BSD sockets compatible API* is implemented. Both blocking and non-blocking datagram (UDP) and stream (TCP) sockets are supported.
- **Secure Sockets API** Experimental support for TLS/DTLS secure protocols and configuration options for sockets API. Secure functions for the implementation are provided by mbedTLS library.
- **MQTT** Message Queue Telemetry Transport (ISO/IEC PRF 20922) is supported. A sample mqtt-publisher-sample client application for MQTT v3.1.1 is implemented.
- **CoAP** Constrained Application Protocol (RFC 7252) is supported. Both coap-client-sample and coap-server-sample sample applications are implemented.
- **LWM2M** OMA Lightweight Machine-to-Machine Protocol (LwM2M specification 1.0.2) is supported via the "Bootstrap", "Client Registration", "Device Management & Service Enablement" and "Information Reporting" interfaces. The required core LwM2M objects are implemented as well as several IPSO Smart Objects. lwm2m-client-sample implements the library as an example.
- **DNS** Domain Name Service (RFC 1035) client functionality is supported. Applications can use the DNS API to query domain name information or IP addresses from the DNS server. Both IPv4 (A) and IPv6 (AAAA) records can be queried. Both multicast DNS (mDNS) (RFC 6762) and link-local multicast name resolution (LLMNR) (RFC 4795) are supported.
- **Network Management API.** Applications can use network management API to listen management events generated by core stack when for example IP address is added to the device, or network interface is coming up etc.
- **Multiple Network Technologies.** The Zephyr OS can be configured to support multiple network technologies at the same time simply by enabling them in Kconfig: for example, Ethernet and 802.15.4 support. Note that no automatic IP routing functionality is provided between these technologies. Applications can send data according to their needs to desired network interface.
- **Minimal Copy Network Buffer Management.** It is possible to have minimal copy network data path. This means that the system tries to avoid copying application data when it is sent to the network.
- **Virtual LAN support.** Virtual LANs (VLANs) allow partitioning of physical ethernet networks into logical networks. See *VLAN support* for more details.
- **Network traffic classification.** The sent and received network packets can be prioritized depending on application needs. See *traffic classification* for more details.
- **Time Sensitive Networking.** The gPTP (generalized Precision Time Protocol) is supported. See *gPTP support* for more details.

- **Network shell.** The network shell provides helpers for figuring out network status, enabling/disabling features, and issuing commands like ping or DNS resolving. The net-shell is useful when developing network software. See *network shell* for more details.

Additionally these network technologies (link layers) are supported in Zephyr OS v1.7 and later:

- IEEE 802.15.4
- Bluetooth
- Ethernet
- SLIP (IP over serial line). Used for testing with QEMU. It provides ethernet interface to host system (like Linux) and test applications can be run in Linux host and send network data to Zephyr OS device.

Source Tree Layout

The networking stack source code tree is organized as follows:

`subsys/net/ip/` This is where the IP stack code is located.

`subsys/net/l2/` This is where the IP stack layer 2 code is located. This includes generic support for Bluetooth IPSP adaptation, Ethernet, IEEE 802.15.4 and Wi-Fi.

`subsys/net/lib/` Application-level protocols (DNS, MQTT, etc.) and additional stack components (BSD Sockets, etc.).

`include/net/` Public API header files. These are the header files applications need to include to use IP networking functionality.

`samples/net/` Sample networking code. This is a good reference to get started with network application development.

`tests/net/` Test applications. These applications are used to verify the functionality of the IP stack, but are not the best source for sample code (see `samples/net` instead).

8.14.2 Network Stack Architecture

Network Packet Processing Statistics

This page describes how to get information about network packet processing statistics inside network stack.

Network stack contains infrastructure to figure out how long the network packet processing takes either in sending or receiving path. There are two Kconfig options that control this. For transmit (TX) path the option is called `CONFIG_NET_PKT_TXTIME_STATS` and for receive (RX) path the options is called `CONFIG_NET_PKT_RXTIME_STATS`. Note that for TX, all kind of network packet statistics is collected. For RX, only UDP, TCP or raw packet type network packet statistics is collected.

After enabling these options, the *net stats* network shell command will show this information:

```
Avg TX net_pkt (11484) time 67 us
Avg RX net_pkt (11474) time 43 us
```

Note: The values above and below are from emulated qemu_x86 board and UDP traffic

The TX time tells how long it took for network packet from its creation to when it was sent to the network. The RX time tells the time from its creation to when it was passed to the application. The values are in microseconds. The statistics will be collected per traffic class if there are more than one

transmit or receive queues defined in the system. These are controlled by `CONFIG_NET_TC_TX_COUNT` and `CONFIG_NET_TC_RX_COUNT` options.

If you enable `CONFIG_NET_PKT_TXTIME_STATS_DETAIL` or `CONFIG_NET_PKT_RXTIME_STATS_DETAIL` options, then additional information for TX or RX network packets are collected when the network packet traverses the IP stack.

After enabling these options, the *net stats* will show this information:

```
Avg TX net_pkt (18902) time 63 us    [0->22->15->23=60 us]
Avg RX net_pkt (18892) time 42 us    [0->9->6->11->13=39 us]
```

The numbers inside the brackets contain information how many microseconds it took for a network packet to go from previous state to next.

In the TX example above, the values are averages over **18902** packets and contain this information:

- Packet was created by application so the time is **0**.
- Packet is about to be placed to transmit queue. The time it took from network packet creation to this state, is **22** microseconds in this example.
- The correct TX thread is invoked, and the packet is read from the transmit queue. It took **15** microseconds from previous state.
- The network packet was just sent and the network stack is about to free the network packet. It took **23** microseconds from previous state.
- In total it took on average **60** microseconds to get the network packet sent. The value **63** tells also the same information, but is calculated differently so there is slight difference because of rounding errors.

In the RX example above, the values are averages over **18892** packets and contain this information:

- Packet was created network device driver so the time is **0**.
- Packet is about to be placed to receive queue. The time it took from network packet creation to this state, is **9** microseconds in this example.
- The correct RX thread is invoked, and the packet is read from the receive queue. It took **6** microseconds from previous state.
- The network packet is then processed and placed to correct socket queue. It took **11** microseconds from previous state.
- The last value tells how long it took from there to the application. Here the value is **13** microseconds.
- In total it took on average **39** microseconds to get the network packet sent. The value **42** tells also the same information, but is calculated differently so there is slight difference because of rounding errors.

The Zephyr network stack is a native network stack specifically designed for Zephyr OS. It consists of layers, each meant to provide certain services to other layers. Network stack functionality is highly configurable via Kconfig options.

High level overview of the network stack

The network stack is layered and consists of the following parts:

- **Network Application.** The network application can either use the provided application-level protocol libraries or access the *BSD socket API* directly to create a network connection, send or receive data, and close a connection. The application can also use the *network management API* to configure the network and set related parameters such as network link options, starting a scan (when applicable), listen network configuration events, etc. The *network interface API* can be used to set IP address to a network interface, taking the network interface down, etc.
- **Network Protocols.** This provides implementations for various protocols such as
  - Application-level network protocols like CoAP, LWM2M, and MQTT. See *application protocols chapter* for information about them.
  - Core network protocols like IPv6, IPv4, UDP, TCP, ICMPv4, and ICMPv6. You access these protocols by using the *BSD socket API*.
- **Network Interface Abstraction.** This provides functionality that is common in all the network interfaces, such as setting network interface down, etc. There can be multiple network interfaces in the system. See *network interface overview* for more details.
- **L2 Network Technologies.** This provides a common API for sending and receiving data to and from an actual network device. See *L2 overview* for more details. These network technologies include *Ethernet*, *IEEE 802.15.4*, *Bluetooth*, *CANBUS*, etc. Some of these technologies support IPv6 header compression (6Lo), see RFC 6282 for details. For example ARP for IPv4 is done by the *Ethernet component*.
- **Network Device Drivers.** The actual low-level device drivers handle the physical sending or receiving of network packets.

Network data flow

An application typically consists of one or more *threads* that execute the application logic. When using the *BSD socket API*, the following things will happen.

Data receiving (RX)

1. A network data packet is received by a device driver.
2. The device driver allocates enough network buffers to store the received data. The network packet is placed in the proper RX queue (implemented by *k_fifo*). By default there is only one receive queue in the system, but it is possible to have up to 8 receive queues. These queues will process incoming packets with different priority. See *Traffic Classification* for more details. The receive queues also act as a way to separate the data processing pipeline (bottom-half) as the device driver is running in an interrupt context and it must do its processing as fast as possible.
3. The network packet is then passed to the correct L2 driver. The L2 driver can check if the packet is proper and modify it if needed, e.g. strip L2 header and frame check sequence, etc.
4. The packet is processed by a network interface. The network statistics are collected if enabled by `CONFIG_NET_STATISTICS`.
5. The packet is then passed to L3 processing. If the packet is IP based, then the L3 layer checks if the packet is a proper IPv6 or IPv4 packet.
6. A socket handler then finds an active socket to which the network packet belongs and puts it in a queue for that socket, in order to separate the networking code from the application. Typically the application is run in userspace context and the network stack is run in kernel context.
7. The application will then receive the data and can process it as needed. The application should have used the *BSD socket API* to create a socket that will receive the data.

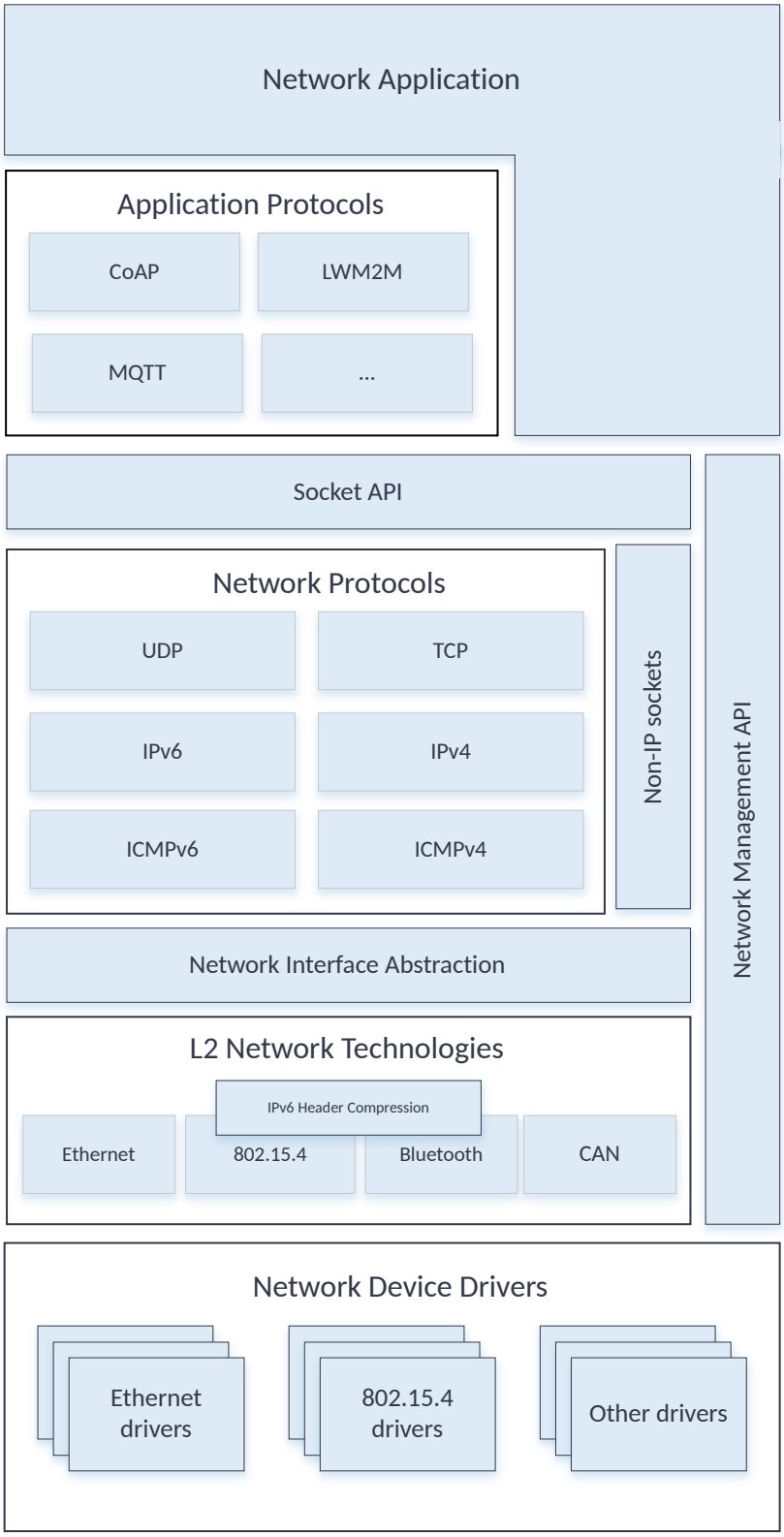


Fig. 8: Network stack overview

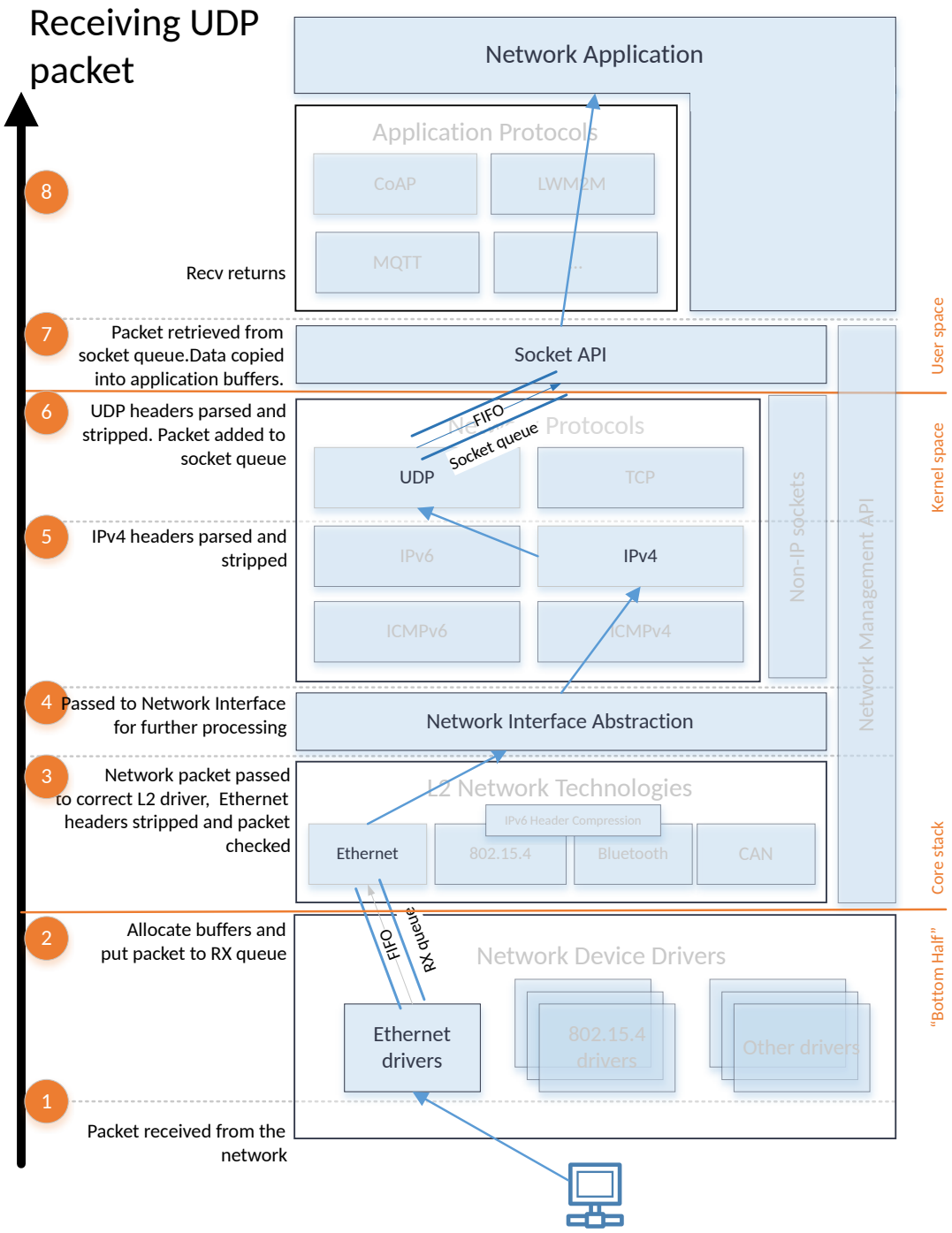


Fig. 9: Network RX data flow

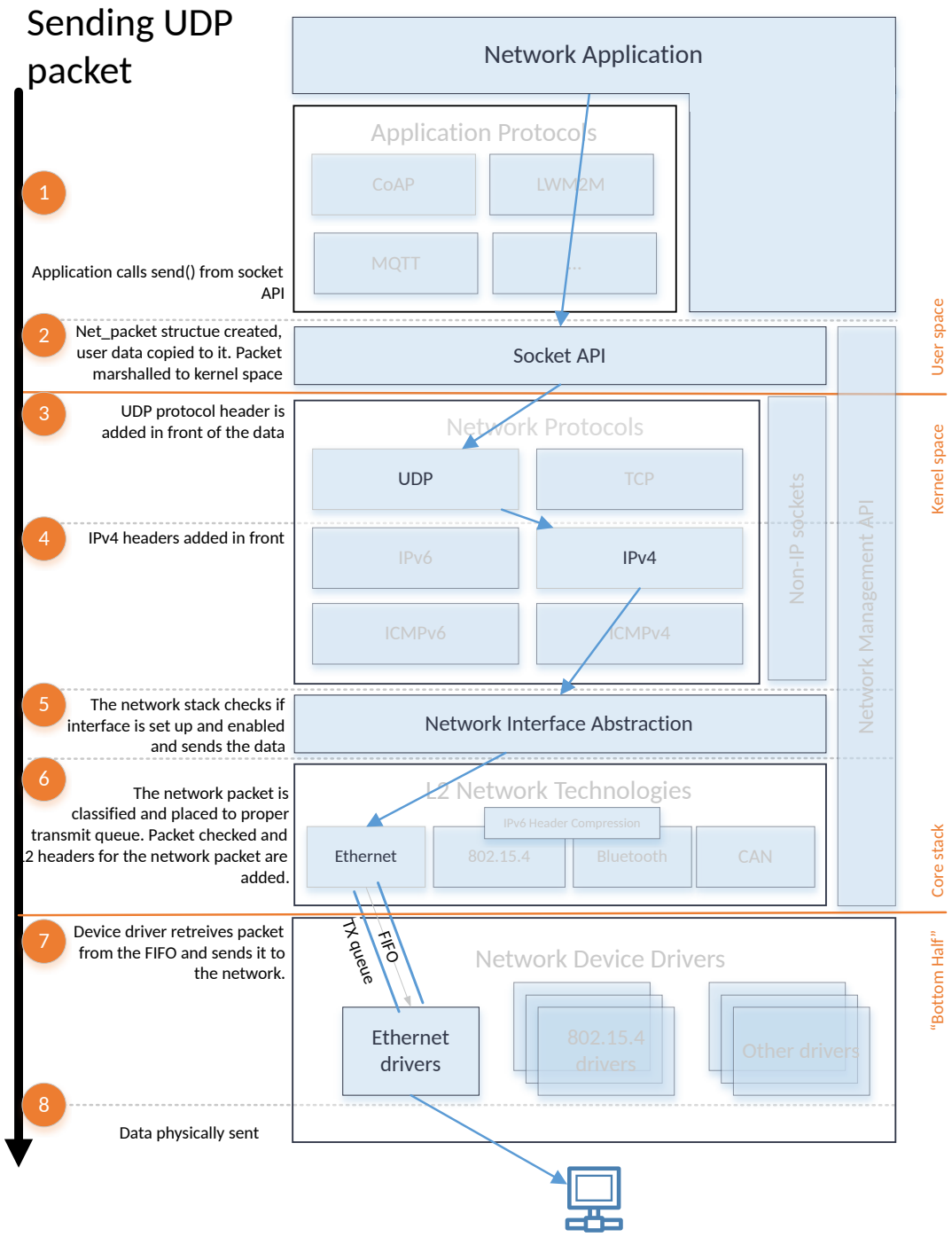


Fig. 10: Network TX data flow

Data sending (TX)

1. The application should use the *BSD socket API* when sending the data.
2. The application data is prepared for sending to kernel space and then copied to internal net_buf structures.
3. Depending on the socket type, a protocol header is added in front of the data. For example, if the socket is a UDP socket, then a UDP header is constructed and placed in front of the data.
4. An IP header is added to the network packet for a UDP or TCP packet.
5. The network stack will check that the network interface is properly set for the network packet, and also will make sure that the network interface is enabled before the data is queued to be sent.
6. The network packet is then classified and placed to the proper transmit queue (implemented by *k_fifo*). By default there is only one transmit queue in the system, but it is possible to have up to 8 transmit queues. These queues will process the sent packets with different priority. See *Traffic Classification* for more details. After the transmit packet classification, the packet is checked by the correct L2 layer module. The L2 module will do additional checks for the data and it will also create any L2 headers for the network packet. If everything is ok, the data is given to the network device driver to be sent out.
7. The device driver will send the packet to the network.

Note that in both the TX and RX data paths, the queues (*k_fifo's*) form separation points where data is passed from one *thread* to another. These *threads* might run in different contexts (*kernel* vs. *userspace*) and with different *priorities*.

Network packet processing statistics

See information about network processing statistics *here*.

8.14.3 Network Connectivity API

Applications should use the BSD socket API defined in include/net/socket.h to create a connection, send or receive data, and close a connection. The same API can be used when working with UDP or TCP data. See *BSD socket API* for more details.

See sockets-echo-server-sample and sockets-echo-client-sample applications how to create a simple server or client BSD socket based application.

The legacy connectivity API in include/net/net_context.h should not be used by applications.

8.14.4 Networking with the host system

Networking with native_posix board

This page describes how to set up a virtual network between a (Linux) host and a Zephyr application running in a native_posix board.

In this example, the sockets-echo-server-sample sample application from the Zephyr source distribution is run in native_posix board. The Zephyr native_posix board instance is connected to a Linux host using a tuntap device which is modeled in Linux as an Ethernet network interface.

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need three terminal windows:

- Terminal #1 is terminal window with net-tools being the current directory (`cd net-tools`)
- Terminal #2 is your usual Zephyr development terminal, with the Zephyr environment initialized.
- Terminal #3 is the console to the running Zephyr native_posix instance (optional).

Step 1 - Create Ethernet interface Before starting native_posix with network emulation, a network interface should be created.

In terminal #1, type:

```
./net-setup.sh
```

You can tweak the behavior of the net-setup.sh script. See various options by running `net-setup.sh` like this:

```
./net-setup.sh --help
```

Step 2 - Start app in native_posix board Build and start the `echo_server` sample application.

In terminal #2, type:

```
west build -b native_posix samples/net/sockets/echo_server
west build -t run
```

Step 3 - Connect to console (optional) The console window should be launched automatically when the Zephyr instance is started but if it does not show up, you can manually connect to the console. The native_posix board will print a string like this when it starts:

```
UART connected to pseudotty: /dev/pts/5
```

You can manually connect to it like this:

```
screen /dev/pts/5
```

Networking with QEMU Ethernet

This page describes how to set up a virtual network between a (Linux) host and a Zephyr application running in QEMU.

In this example, the sockets-echo-server-sample sample application from the Zephyr source distribution is run in QEMU. The Zephyr instance is connected to a Linux host using a tuntap device which is modeled in Linux as an Ethernet network interface.

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need two terminal windows:

- Terminal #1 is terminal window with net-tools being the current directory (`cd net-tools`)
- Terminal #2 is your usual Zephyr development terminal, with the Zephyr environment initialized.

When configuring the Zephyr instance, you must select the correct Ethernet driver for QEMU connectivity:

- For `qemu_x86`, select `Intel(R) PRO/1000 Gigabit Ethernet driver` Ethernet driver. Driver is called `e1000` in Zephyr source tree.
- For `qemu_cortex_m3`, select `TI Stellaris MCU family ethernet driver` Ethernet driver. Driver is called `stellaris` in Zephyr source tree.
- For `mps2_an385`, select `SMSC911x/9220 Ethernet driver` Ethernet driver. Driver is called `smsc911x` in Zephyr source tree.

Step 1 - Create Ethernet interface Before starting QEMU with network connectivity, a network interface should be created in the host system.

In terminal #1, type:

```
./net-setup.sh
```

You can tweak the behavior of the `net-setup.sh` script. See various options by running `net-setup.sh` like this:

```
./net-setup.sh --help
```

Step 2 - Start app in QEMU board Build and start the sockets-echo-server-sample sample application. In this example, the qemu_x86 board is used.

In terminal #2, type:

```
west build -b qemu_x86 samples/net/sockets/echo_server -- -DOVERLAY_CONFIG=overlay-
↪e1000.conf
west build -t run
```

Exit QEMU by pressing `CTRL+A x`.

Networking with QEMU

This page describes how to set up a virtual network between a (Linux) host and a Zephyr application running in a QEMU virtual machine (built for Zephyr targets such as qemu_x86 and qemu_cortex_m3).

In this example, the sockets-echo-server-sample sample application from the Zephyr source distribution is run in QEMU. The QEMU instance is connected to a Linux host using a serial port, and SLIP is used to transfer data between the Zephyr application and Linux (over a chain of virtual connections).

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
cd net-tools
make
```

Note: If you get an error about AX_CHECK_COMPILE_FLAG, install package `autoconf-archive` package on Debian/Ubuntu.

Basic Setup For the steps below, you will need at least 4 terminal windows:

- Terminal #1 is your usual Zephyr development terminal, with the Zephyr environment initialized.
- Terminals #2, #3, and #4 are terminal windows with net-tools being the current directory (`cd net-tools`)

Step 1 - Create helper socket Before starting QEMU with network emulation, a Unix socket for the emulation should be created.

In terminal #2, type:

```
./loop-socat.sh
```

Step 2 - Start TAP device routing daemon In terminal #3, type:

```
sudo ./loop-slip-tap.sh
```

For applications requiring DNS, you may need to restart the host's DNS server at this point, as described in *Setting up Zephyr and NAT/masquerading on host to access Internet*.

Step 3 - Start app in QEMU Build and start the `echo_server` sample application.

In terminal #1, type:

```
west build -b qemu_x86 samples/net/sockets/echo_server
west build -t run
```

If you see an error from QEMU about unix:/tmp/slip.sock, it means you missed Step 1 above.

Step 4 - Run apps on host Now in terminal #4, you can run various tools to communicate with the application running in QEMU.

You can start with pings:

```
ping 192.0.2.1
ping6 2001:db8::1
```

You can use the netcat ("nc") utility, connecting using UDP:

```
echo foobar | nc -6 -u 2001:db8::1 4242
foobar
```

```
echo foobar | nc -u 192.0.2.1 4242
foobar
```

If echo_server is compiled with TCP support (now enabled by default for the echo_server sample, CONFIG_NET_TCP=y):

```
echo foobar | nc -6 -q2 2001:db8::1 4242
foobar
```

Note: Use Ctrl+C to exit.

You can also use the telnet command to achieve the above.

Step 5 - Stop supporting daemons When you are finished with network testing using QEMU, you should stop any daemons or helpers started in the initial steps, to avoid possible networking or routing problems such as address conflicts in local network interfaces. For example, stop them if you switch from testing networking with QEMU to using real hardware, or to return your host laptop to normal Wi-Fi use.

To stop the daemons, press Ctrl+C in the corresponding terminal windows (you need to stop both `loop-slip-tap.sh` and `loop-socat.sh`).

Exit QEMU by pressing `CTRL+A x`.

Setting up Zephyr and NAT/masquerading on host to access Internet To access the internet from a Zephyr application, some additional setup on the host may be required. This setup is common for both application running in QEMU and on real hardware, assuming that a development board is connected to the development host. If a board is connected to a dedicated router, it should not be needed.

To access the internet from a Zephyr application using IPv4, a gateway should be set via DHCP or configured manually. For applications using the "Settings" facility (with the config option CONFIG_NET_CONFIG_SETTINGS enabled), set the CONFIG_NET_CONFIG_MY_IPV4_GW option to the IP address of the gateway. For apps not using the "Settings" facility, set up the gateway by calling the *net_if_ipv4_set_gw()* at runtime.

To access the internet from a custom application running in QEMU, NAT (masquerading) should be set up for QEMU's source address. Assuming 192.0.2.1 is used, the following command should be run as root:

```
iptables -t nat -A POSTROUTING -j MASQUERADE -s 192.0.2.1
```

Additionally, IPv4 forwarding should be enabled on the host, and you may need to check that other firewall (iptables) rules don't interfere with masquerading. To enable IPv4 forwarding the following command should be run as root:

```
sysctl -w net.ipv4.ip_forward=1
```

Some applications may also require a DNS server. A number of Zephyr-provided samples assume by default that the DNS server is available on the host (IP 192.0.2.2), which, in modern Linux distributions, usually runs at least a DNS proxy. When running with QEMU, it may be required to restart the host's DNS, so it can serve requests on the newly created TAP interface. For example, on Debian-based systems:

```
service dnsmasq restart
```

An alternative to relying on the host's DNS server is to use one in the network. For example, 8.8.8.8 is a publicly available DNS server. You can configure it using CONFIG_DNS_SERVER1 option.

Network connection between two QEMU VMs Unlike the VM-to-Host setup described above, VM-to-VM setup is automatic. For sample applications that support this mode (such as the echo_server and echo_client samples), you will need two terminal windows, set up for Zephyr development.

Terminal #1:

```
west build -b qemu_x86 samples/net/sockets/echo_server
```

This will start QEMU, waiting for a connection from a client QEMU.

Terminal #2:

```
west build -b qemu_x86 samples/net/sockets/echo_client
```

This will start a second QEMU instance, where you should see logging of data sent and received in both.

Running multiple QEMU VMs of the same sample If you find yourself wanting to run multiple instances of the same Zephyr sample application, which do not need to talk to each other, use the QEMU_INSTANCE argument.

Start socat and tunslip6 manually (instead of using the loop-xxx.sh scripts) for as many instances as you want. Use the following as a guide, replacing MAIN or OTHER.

Terminal #1:

```
socat PTY,link=/tmp/slip.devMAIN UNIX-LISTEN:/tmp/slip.sockMAIN
$ZEPHYR_BASE/../net-tools/tunslip6 -t tapMAIN -T -s /tmp/slip.devMAIN \
     2001:db8::1/64
# Now run Zephyr
make -Cbuild run QEMU_INSTANCE=MAIN
```

Terminal #2:

```
socat PTY,link=/tmp/slip.devOTHER UNIX-LISTEN:/tmp/slip.sockOTHER
$ZEPHYR_BASE/../net-tools/tunslip6 -t tapOTHER -T -s /tmp/slip.devOTHER \
     2001:db8::1/64
make -Cbuild run QEMU_INSTANCE=OTHER
```

USB Device Networking

This page describes how to set up networking between a Linux host and a Zephyr application running on USB supported devices.

The board is connected to Linux host using USB cable and provides an Ethernet interface to the host. The sockets-echo-server-sample application from the Zephyr source distribution is run on supported board. The board is connected to a Linux host using a USB cable providing an Ethernet interface to the host.

Basic Setup To communicate with the Zephyr application over a newly created Ethernet interface, we need to assign IP addresses and set up a routing table for the Linux host. After plugging a USB cable from the board to the Linux host, the `cdc_ether` driver registers a new Ethernet device with a provided MAC address.

You can check that network device is created and MAC address assigned by running dmesg from the Linux host.

```
cdc_ether 1-2.7:1.0 eth0: register 'cdc_ether' at usb-0000:00:01.2-2.7, CDC Ethernet␣
↪Device, 00:00:5e:00:53:01
```

We need to set it up and assign IP addresses as explained in the following section.

Choosing IP addresses To establish network connection to the board we need to choose IP address for the interface on the Linux host.

It make sense to choose addresses in the same subnet we have in Zephyr application. IP addresses usually set in the project configuration files and may be checked also from the shell with following commands. Connect a serial console program (such as puTTY) to the board, and enter this command to the Zephyr shell:

```
shell> net iface

Interface 0xa800e580 (Ethernet)
===============================
Link addr : 00:00:5E:00:53:00
MTU       : 1500
IPv6 unicast addresses (max 2):
        fe80::200:5eff:fe00:5300 autoconf preferred infinite
        2001:db8::1 manual preferred infinite
...
```

(continues on next page)

(continued from previous page)

```
IPv4 unicast addresses (max 1):
        192.0.2.1 manual preferred infinite
```

This command shows that one IPv4 address and two IPv6 addresses have been assigned to the board. We can use either IPv4 or IPv6 for network connection depending on the board network configuration.

Next step is to assign IP addresses to the new Linux host interface, in the following steps `enx00005e005301` is the name of the interface on my Linux system.

Setting IPv4 address and routing

```
# ip address add dev enx00005e005301 192.0.2.2
# ip link set enx00005e005301 up
# ip route add 192.0.2.0/24 dev enx00005e005301
```

Setting IPv6 address and routing

```
# ip address add dev enx00005e005301 2001:db8::2
# ip link set enx00005e005301 up
# ip -6 route add 2001:db8::/64 dev enx00005e005301
```

Testing connection From the host we can test the connection by pinging Zephyr IP address of the board with:

```
$ ping 192.0.2.1
PING 192.0.2.1 (192.0.2.1) 56(84) bytes of data.
64 bytes from 192.0.2.1: icmp_seq=1 ttl=64 time=2.30 ms
64 bytes from 192.0.2.1: icmp_seq=2 ttl=64 time=1.43 ms
64 bytes from 192.0.2.1: icmp_seq=3 ttl=64 time=2.45 ms
...
```

Networking with QEMU User

This page is intended to serve as a starting point for anyone interested in using QEMU SLIRP with Zephyr.

Introduction SLIRP is a network backend which provides the complete TCP/IP stack within QEMU and uses that stack to implement a virtual NAT'd network. As there are no dependencies on the host, SLIRP is simple to setup.

By default, QEMU uses the `10.0.2.X/24` network and runs a gateway at `10.0.2.2`. All traffic intended for the host network has to travel through this gateway, which will filter out packets based on the QEMU command line parameters. This gateway also functions as a DHCP server for all GOS, allowing them to be automatically assigned with an IP address starting from `10.0.2.15`.

More details about User Networking can be obtained from here: https://wiki.qemu.org/Documentation/Networking#User_Networking_.28SLIRP.29

Using SLIRP with Zephyr In order to use SLIRP with Zephyr, the user has to set the Kconfig option to enable User Networking.

```
CONFIG_NET_QEMU_USER=y
```

Once this configuration option is enabled, all QEMU launches will use SLIRP. In the default configuration, Zephyr only enables User Networking, and does not pass any arguments to it. This means that the Guest will only be able to communicate to the QEMU gateway, and any data intended for the host machine will be dropped by QEMU.

In general, QEMU User Networking can take in a lot of arguments including,

- Information about host/guest port forwarding. This must be provided to create a communication channel between the guest and host.
- Information about network to use. This may be valuable if the user does not want to use the default `10.0.2.X` network.
- Tell QEMU to start DHCP server at user-defined IP address.
- ID and other information.

As this information varies with every use case, it is difficult to come up with good defaults that work for all. Therefore, Zephyr Implementation offloads this to the user, and expects that they will provide arguments based on requirements. For this, there is a Kconfig string which can be populated by the user.

```
CONFIG_NET_QEMU_USER_EXTRA_ARGS="net=192.168.0.0/24,hostfwd=tcp::8080-:8080"
```

This option is appended as-is to the QEMU command line. Therefore, any problems with this command line will be reported by QEMU only. Here's what this particular example will do,

- Make QEMU use the `192.168.0.0/24` network instead of the default.
- Enable forwarding of any TCP data received from port 8080 of host to port 8080 of guest, and vice versa.

Limitations If the user does not have any specific networking requirements other than the ability to access a web page from the guest, user networking (slirp) is a good choice. However, it has several limitations

- There is a lot of overhead so the performance is poor.
- The guest is not directly accessible from the host or the external network.
- In general, ICMP traffic does not work (so you cannot use ping within a guest).
- As port mappings need to be defined before launching qemu, clients which use dynamically generated ports cannot communicate with external network.
- There is a bug in the SLIRP implementation which filters out all IPv6 packets from the guest. See https://bugs.launchpad.net/qemu/+bug/1724590 for details. Therefore, IPv6 will not work with User Networking.

Networking with multiple Zephyr instances

– *Step 4 - Start Zephyr instances*

This page describes how to set up a virtual network between multiple Zephyr instances. The Zephyr instances could be running inside QEMU or could be native_posix board processes. The Linux host can be used to route network traffic between these systems.

Prerequisites On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need five terminal windows:

- Terminal #1 and #2 are terminal windows with net-tools being the current directory (`cd net-tools`)
- Terminal #3, where you setup bridging in Linux host
- Terminal #4 and #5 are your usual Zephyr development terminal, with the Zephyr environment initialized.

As there are multiple ways to setup the Zephyr network, the example below uses `qemu_x86` board with `e1000` Ethernet controller and native_posix board to simplify the setup instructions. You can use other QEMU boards and drivers if needed, see *Networking with QEMU Ethernet* for details. You can also use two or more native_posix board Zephyr instances and connect them together.

Step 1 - Create configuration files Before starting QEMU with network connectivity, a network interfaces for each Zephyr instance should be created in the host system. The default setup for creating network interface cannot be used here as that is for connecting one Zephyr instance to Linux host.

For Zephyr instance #1, create file called `zephyr1.conf` to `net-tools` project, or to some other suitable directory.

```
# Configuration file for setting IP addresses for a network interface.
INTERFACE="$1"
HWADDR="00:00:5e:00:53:11"
IPV6_ADDR_1="2001:db8:100::2"
IPV6_ROUTE_1="2001:db8:100::/64"
IPV4_ADDR_1="198.51.100.2/24"
IPV4_ROUTE_1="198.51.100.0/24"
ip link set dev $INTERFACE up
ip link set dev $INTERFACE address $HWADDR
ip -6 address add $IPV6_ADDR_1 dev $INTERFACE nodad
ip -6 route add $IPV6_ROUTE_1 dev $INTERFACE
ip address add $IPV4_ADDR_1 dev $INTERFACE
ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
```

For Zephyr instance #2, create file called `zephyr2.conf` to `net-tools` project, or to some other suitable directory.

```
# Configuration file for setting IP addresses for a network interface.
INTERFACE="$1"
HWADDR="00:00:5e:00:53:22"
IPV6_ADDR_1="2001:db8:200::2"
IPV6_ROUTE_1="2001:db8:200::/64"
IPV4_ADDR_1="203.0.113.2/24"
IPV4_ROUTE_1="203.0.113.0/24"
```

(continues on next page)

(continued from previous page)

```
ip link set dev $INTERFACE up
ip link set dev $INTERFACE address $HWADDR
ip -6 address add $IPV6_ADDR_1 dev $INTERFACE nodad
ip -6 route add $IPV6_ROUTE_1 dev $INTERFACE
ip address add $IPV4_ADDR_1 dev $INTERFACE
ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
```

Step 2 - Create Ethernet interfaces The following `net-setup.sh` commands should be typed in net-tools directory (`cd net-tools`).

In terminal #1, type:

```
./net-setup.sh -c zephyr1.conf -i zeth.1
```

In terminal #2, type:

```
./net-setup.sh -c zephyr2.conf -i zeth.2
```

Step 3 - Setup network bridging In terminal #3, type:

```
sudo brctl addbr zeth-br
sudo brctl addif zeth-br zeth.1
sudo brctl addif zeth-br zeth.2
sudo ifconfig zeth-br up
```

Step 4 - Start Zephyr instances In this example we start sockets-echo-server-sample and sockets-echo-client-sample applications. You can use other applications too as needed.

In terminal #4, if you are using QEMU, type this:

```
west build -d build/server -b qemu_x86 -t run \
   samples/net/sockets/echo_server -- \
   -DOVERLAY_CONFIG=overlay-e1000.conf \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"203.0.113.1\" \
   -DCONFIG_ETH_QEMU_IFACE_NAME=\"zeth.1\" \
   -DCONFIG_ETH_QEMU_EXTRA_ARGS=\"mac=00:00:5e:00:53:01\"
```

or if you want to use native_posix board, type this:

```
west build -d build/server -b native_posix -t run \
   samples/net/sockets/echo_server -- \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"203.0.113.1\" \
   -DCONFIG_ETH_NATIVE_POSIX_DRV_NAME=\"zeth.1\" \
   -DCONFIG_ETH_NATIVE_POSIX_MAC_ADDR=\"00:00:5e:00:53:01\" \
   -DCONFIG_ETH_NATIVE_POSIX_RANDOM_MAC=n
```

In terminal #5, if you are using QEMU, type this:

```
west build -d build/client -b qemu_x86 -t run \
   samples/net/sockets/echo_client -- \
   -DOVERLAY_CONFIG=overlay-e1000.conf \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"198.51.100.1\" \
   -DCONFIG_ETH_QEMU_IFACE_NAME=\"zeth.2\" \
   -DCONFIG_ETH_QEMU_EXTRA_ARGS=\"mac=00:00:5e:00:53:02\"
```

or if you want to use native_posix board, type this:

```
west build -d build/client -b native_posix -t run \
   samples/net/sockets/echo_client -- \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"198.51.100.1\" \
   -DCONFIG_ETH_NATIVE_POSIX_DRV_NAME=\"zeth.2\" \
   -DCONFIG_ETH_NATIVE_POSIX_MAC_ADDR=\"00:00:5e:00:53:02\" \
   -DCONFIG_ETH_NATIVE_POSIX_RANDOM_MAC=n
```

Also if you have firewall enabled in your host, you need to allow traffic between `zeth.1`, `zeth.2` and `zeth-br` interfaces.

Networking with QEMU and IEEE 802.15.4

This page describes how to set up a virtual network between two QEMUs that are connected together via UART and are running IEEE 802.15.4 link layer between them. Note that this only works in Linux host.

Basic Setup For the steps below, you will need two terminal windows:

- Terminal #1 is terminal window with `echo-server` Zephyr sample application.
- Terminal #2 is terminal window with `echo-client` Zephyr sample application.

If you want to capture the transferred network data, you must compile the `monitor_15_4` program in `net-tools` directory.

Open a terminal window and type:

```
cd $ZEPHYR_BASE/../net-tools
make monitor_15_4
```

Step 1 - Compile and start echo-server In terminal #1, type:

```
west build -b qemu_x86 -d build/server samples/net/sockets/echo_server -- -DOVERLAY_
↪CONFIG=overlay-qemu_802154.conf
west build -t server -d build/server
```

If you want to capture the network traffic between the two QEMUs, type:

```
west build -b qemu_x86 -d build/server samples/net/sockets/echo_server -- -G'Unix␣
↪Makefiles' -DOVERLAY_CONFIG=overlay-qemu_802154.conf -DPCAP=capture.pcap
west build -t server -d build/server
```

Note that the `make` must be used for `server` target if packet capture option is set in command line. The `build/server/capture.pcap` file will contain the transferred data.

Step 2 - Compile and start echo-client In terminal #2, type:

```
west build -b qemu_x86 -d build/client samples/net/sockets/echo_client -- -DOVERLAY_
↪CONFIG=overlay-qemu_802154.conf
west build -t client -d build/client
```

You should see data passed between the two QEMUs. Exit QEMU by pressing `CTRL+A x`.

While developing networking software, it is usually necessary to connect and exchange data with the host system like a Linux desktop computer. Depending on what board is used for development, the following options are possible:

- QEMU using SLIP (Serial Line Internet Protocol).
  - Here IP packets are exchanged between Zephyr and the host system via serial port. This is the legacy way of transferring data. It is also quite slow so use it only when necessary. See *Networking with QEMU* for details.
- QEMU using built-in Ethernet driver.
  - Here IP packets are exchanged between Zephyr and the host system via QEMU's built-in Ethernet driver. Not all QEMU boards support built-in Ethernet so in some cases, you might need to use the SLIP method for host connectivity. See *Networking with QEMU Ethernet* for details.
- QEMU using SLIRP (Qemu User Networking).
  - QEMU User Networking is implemented using "slirp", which provides a full TCP/IP stack within QEMU and uses that stack to implement a virtual NAT'd network. As this support is built into QEMU, it can be used with any model and requires no admin privileges on the host machine, unlike TAP. However, it has several limitations including performance which makes it less valuable for practical purposes. See *Networking with QEMU User* for details.
- native_posix board.
  - The Zephyr instance can be executed as a user space process in the host system. This is the most convenient way to debug the Zephyr system as one can attach host debugger directly to the running Zephyr instance. This requires that there is an adaptation driver in Zephyr for interfacing with the host system. An Ethernet driver exists in Zephyr for this purpose. See *Networking with native_posix board* for details.
- USB device networking.
  - Here, the Zephyr instance is run on a real board and the connectivity to the host system is done via USB. See *USB Device Networking* for details.
- Connecting multiple Zephyr instances together.
  - If you have multiple Zephyr instances, either QEMU or native_posix ones, and want to create a connection between them, see *Networking with multiple Zephyr instances* for details.
- Simulating IEEE 802.15.4 network between two QEMUs.
  - Here, two Zephyr instances are running and there is IEEE 802.15.4 link layer run over an UART between them. See *Networking with QEMU and IEEE 802.15.4* for details.

8.14.5 Monitor Network Traffic

It is useful to be able to monitor the network traffic especially when debugging a connectivity issues or when developing new protocol support in Zephyr. This page describes how to set up a way to capture network traffic so that user is able to use Wireshark or similar tool in remote host to see the network packets sent or received by a Zephyr device.

See also the net-capture-sample sample application from the Zephyr source distribution for configuration options that need to be enabled.

Host Configuration

The instructions here describe how to setup a Linux host to capture Zephyr network RX and TX traffic. Similar instructions should work also in other operating systems. On the Linux Host, fetch the Zephyr `net-tools` project, which is located in a separate Git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

The `net-tools` project provides a configure file to setup IP-to-IP tunnel interface so that we can transfer monitoring data from Zephyr to host.

In terminal #1, type:

```
./net-setup.sh -c zeth-tunnel.conf
```

This script will create following IPIP tunnel interfaces:

Interface name	Description
`zeth-ip6ip`	IPv6-over-IPv4 tunnel
`zeth-ipip`	IPv4-over-IPv4 tunnel
`zeth-ipip6`	IPv4-over-IPv6 tunnel
`zeth-ip6ip6`	IPv6-over-IPv6 tunnel

Zephyr will send captured network packets to one of these interfaces. The actual interface will depend on how the capturing is configured. You can then use Wireshark to monitor the proper network interface.

After the tunneling interfaces have been created, you can use for example `net-capture.py` script from `net-tools` project to print or save the captured network packets. The `net-capture.py` provides an UDP listener, it can print the captured data to screen and optionally can also save the data to a pcap file.

```
$ ./net-capture.py -i zeth-ip6ip -w capture.pcap
[20210408Z14:33:08.959589] Ether / IP / ICMP 192.0.2.1 > 192.0.2.2 echo-request 0 /␣
↪Raw
[20210408Z14:33:08.976178] Ether / IP / ICMP 192.0.2.2 > 192.0.2.1 echo-reply 0 / Raw
[20210408Z14:33:16.176303] Ether / IPv6 / ICMPv6 Echo Request (id: 0x9feb seq: 0x0)
[20210408Z14:33:16.195326] Ether / IPv6 / ICMPv6 Echo Reply (id: 0x9feb seq: 0x0)
[20210408Z14:33:21.194979] Ether / IPv6 / ICMPv6ND_NS / ICMPv6 Neighbor Discovery␣
↪Option - Source Link-Layer Address 02:00:5e:00:53:3b
[20210408Z14:33:21.217528] Ether / IPv6 / ICMPv6ND_NA / ICMPv6 Neighbor Discovery␣
↪Option - Destination Link-Layer Address 00:00:5e:00:53:ff
```

(continues on next page)

(continued from previous page)

```
[20210408Z14:34:10.245408] Ether / IPv6 / UDP 2001:db8::2:47319 > 2001:db8::1:4242 /␣
↪Raw
[20210408Z14:34:10.266542] Ether / IPv6 / UDP 2001:db8::1:4242 > 2001:db8::2:47319 /␣
↪Raw
```

The `net-capture.py` has following command line options:

```
Listen captured network data from Zephyr and save it optionally to pcap file.
./net-capture.py \
     -i | --interface <network interface>
             Listen this inferface for the data
     [-p | --port <UDP port>]
             UDP port (default is 4242) where the capture data is received
     [-q | --quiet]
             Do not print packet information
     [-t | --type <L2 type of the data>]
             Scapy L2 type name of the UDP payload, default is Ether
     [-w | --write <pcap file name>]
             Write the received data to file in PCAP format
```

Instead of the `net-capture.py` script, you can for example use `netcat` to provide an UDP listener so that the host will not send port unreachable message to Zephyr:

```
nc -l -u 2001:db8:200::2 4242 > /dev/null
```

The IP address above is the inner tunnel endpoint, and can be changed and it depends on how the Zephyr is configured. Zephyr will send UDP packets containing the captured network packets to the configured IP tunnel, so we need to terminate the network connection like this.

Zephyr Configuration

In this example, we use `native_posix` board. You can also use any other board that supports networking.

In terminal #3, type:

```
west build -b native_posix samples/net/capture -- -DCONFIG_NATIVE_UART_AUTOATTACH_
↪DEFAULT_CMD=\""gnome-terminal -- screen %s"\"
```

To see the Zephyr console and shell, start Zephyr instance like this:

```
build/zephyr/zephyr.exe -attach_uart
```

Any other application can be used too, just make sure that suitable configuration options are enabled (see `samples/net/capture/prj.conf` file for examples).

The network capture can be configured automatically if needed, but currently the `capture` sample application does not do that. User has to use `net-shell` to setup and enable the monitoring.

The network packet monitoring needs to be setup first. The `net-shell` has `net capture setup` command for doing that. The command syntax is

```
net capture setup <remote-ip-addr> <local-ip-addr> <peer-ip-addr>
     <remote> is the (outer) endpoint IP address
     <local> is the (inner) local IP address
     <peer> is the (inner) peer IP address
     Local and Peer IP addresses can have UDP port number in them (optional)
     like 198.0.51.2:9000 or [2001:db8:100::2]:4242
```

In Zephyr console, type:

```
net capture setup 192.0.2.2 2001:db8:200::1 2001:db8:200::2
```

This command will create the tunneling interface. The `192.0.2.2` is the remote host where the tunnel is terminated. The address is used to select the local network interface where the tunneling interface is attached to. The `2001:db8:200::1` tells the local IP address for the tunnel, the `2001:db8:200::2` is the peer IP address where the captured network packets are sent. The port numbers for UDP packet can be given in the setup command like this for IPv6-over-IPv4 tunnel

```
net capture setup 192.0.2.2 [2001:db8:200::1]:9999 [2001:db8:200::2]:9998
```

and like this for IPv4-over-IPv4 tunnel

```
net capture setup 192.0.2.2 198.51.100.1:9999 198.51.100.2:9998
```

If the port number is omitted, then `4242` UDP port is used as a default.

The current monitoring configuration can be checked like this:

```
uart:~$ net capture
Network packet capture disabled
                Capture  Tunnel
Device          iface    iface   Local                   Peer
NET_CAPTURE0    -        1      [2001:db8:200::1]:4242  [2001:db8:200::2]:4242
```

which will print the current configuration. As we have not yet enabled monitoring, the `Capture iface` is not set.

Then we need to enable the network packet monitoring like this:

```
net capture enable 2
```

The `2` tells the network interface which traffic we want to capture. In this example, the `2` is the `native_posix` board Ethernet interface. Note that we send the network traffic to the same interface that we are monitoring in this example. The monitoring system avoids to capture already captured network traffic as that would lead to recursion. You can use `net iface` command to see what network interfaces are available. Note that you cannot capture traffic from the tunnel interface as that would cause recursion loop. The captured network traffic can be sent to some other network interface if configured so. Just set the `<remote-ip-addr>` option properly in `net capture setup` so that the IP tunnel is attached to desired network interface. The capture status can be checked again like this:

```
uart:~$ net capture
Network packet capture enabled
                Capture  Tunnel
Device          iface    iface   Local                   Peer
NET_CAPTURE0    2        1      [2001:db8:200::1]:4242  [2001:db8:200::2]:4242
```

After enabling the monitoring, the system will send captured (either received or sent) network packets to the tunnel interface for further processing.

The monitoring can be disabled like this:

```
net capture disable
```

which will turn currently running monitoring off. The monitoring setup can be cleared like this:

```
net capture cleanup
```

It is not necessary to use `net-shell` for configuring the monitoring. The *network capture API* functions can be called by the application if needed.

Wireshark Configuration

The Wireshark tool can be used to monitor the captured network traffic in a useful way.

You can monitor either the tunnel interfaces or the `zeth` interface. In order to see the actual captured data inside an UDP packet, see Wireshark decapsulate UDP document for instructions.

8.15 Using with PlatformIO

8.15.1 What is PlatformIO?

PlatformIO is a cross-platform embedded development environment with Zephyr support maintained by its developers.

Since Zephyr support within PlatformIO is not maintained by the Zephyr Project, please report any issues with PlatformIO directly to its developers in the official PlatformIO repositories.

A detailed overview of the PlatformIO ecosystem and its philosophy can be found in the official PlatformIO documentation.

8.15.2 Installation

- PlatformIO IDE is a toolset for embedded C/C++ development available on Windows, macOS and Linux platforms
- PlatformIO Core (CLI) is a command-line tool that consists of multi-platform build system, platform and library managers and other integration components. It can be used with a variety of code development environments and allows integration with cloud platforms and web services

8.15.3 Configuration

Please go through the official PlatformIO configuration guide for Zephyr project.

8.15.4 Tutorials

- Zephyr and Nordic nRF52-DK: debugging, unit testing, project analysis
- Developing Zephyr RTOS embedded applications on PlatformIO and simulating on Antmicro Renode

8.15.5 Project Examples

Please check the official examples for various development platforms

8.15.6 Next Steps

Here are some useful links for exploring the PlatformIO ecosystem:

- Try other platforms that support Zephyr project
- Learn more about integrations with other IDEs/Text Editors
- Get help from PlatformIO community

8.16 OS Abstraction

OS abstraction layers (OSAL) provide wrapper function APIs that encapsulate common system functions offered by any operating system. These APIs make it easier and quicker to develop for, and port code to multiple software and hardware platforms.

These sections describe the software and hardware abstraction layers supported by the Zephyr RTOS.

8.16.1 POSIX Support

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. Zephyr implements a subset of the embedded profiles PSE51 and PSE52, and BSD Sockets API.

With the POSIX support available in Zephyr, an existing POSIX compliant application can be ported to run on the Zephyr kernel, and therefore leverage Zephyr features and functionality. Additionally, a library designed for use with POSIX threading compatible operating systems can be ported to Zephyr kernel based applications with minimal or no changes.

The POSIX API subset is an increasingly popular OSAL (operating system abstraction layer) for IoT and embedded applications, as can be seen in Zephyr, AWS:FreeRTOS, TI-RTOS, and NuttX.

Benefits of POSIX support in Zephyr include:

- Offering a familiar API to non-embedded programmers, especially from Linux
- Enabling reuse (portability) of existing libraries based on POSIX APIs
- Providing an efficient API subset appropriate for small (MCU) embedded systems

System Overview

Units of Functionality The system profile is defined in terms of component profiles that specify Units of Functionality that can be combined to realize the application platform. A Unit of Functionality is a defined set of services which can be implemented. If implemented, the standard prescribes that all services in the Unit must be implemented.

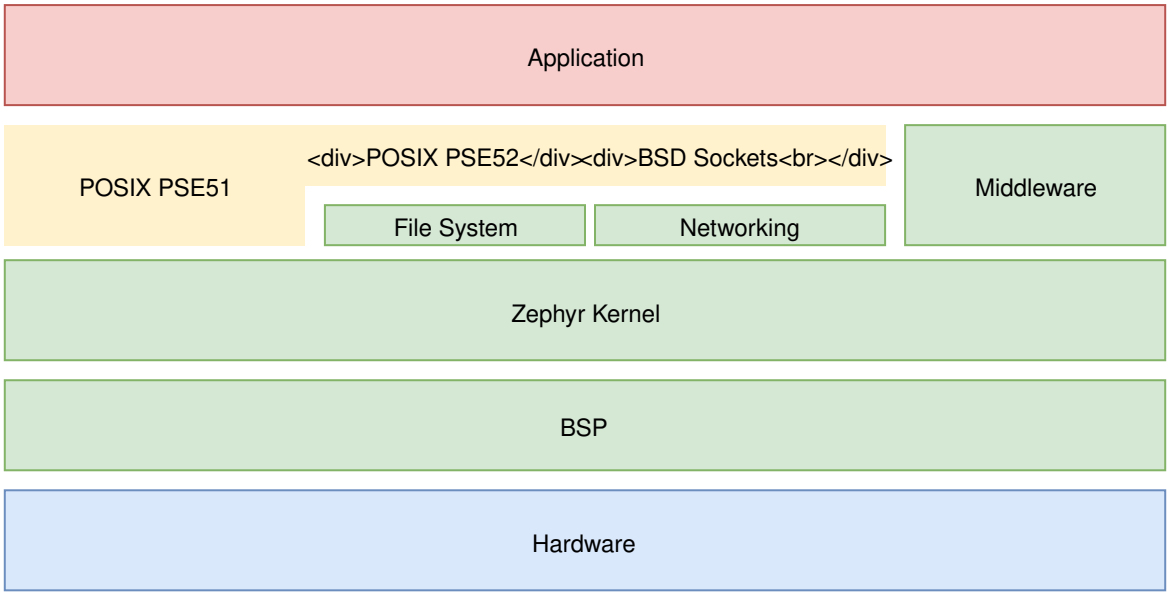


Fig. 11: POSIX support in Zephyr

A Minimal Realtime System Profile implementation must support the following Units of Functionality as defined in IEEE Std. 1003.1 (also referred to as POSIX.1-2017).

Table 7: Units of Functionality

Requirements	Supported	Remarks
POSIX_C_LANG_JUMP		
POSIX_C_LANG_SUPPORT	•	
POSIX_DEVICE_IO		
POSIX_FILE_LOCKING		
POSIX_SIGNALS		
POSIX_SINGLE_PROCESS		
POSIX_THREADS_BASE	•	
XSI_THREAD_MUTEX_EXT	•	
XSI_THREADS_EXT	•	

Option Requirements An implementation supporting the Minimal Realtime System Profile must support the POSIX.1 Option Requirements which are defined in the standard. Options Requirements are used for further sub-profiling within the units of functionality: they further define the functional behavior of the system service (normally adding extra functionality). Depending on the profile to which the POSIX implementation complies,parameters and/or the precise functionality of certain services may differ.

The following list shows the option requirements that are implemented in Zephyr.

Table 8: Option Requirements

Requirements	Supported
_POSIX_CLOCK_SELECTION	
_POSIX_FSYNC	
_POSIX_MEMLOCK	
_POSIX_MEMLOCK_RANGE	
_POSIX_MONOTONIC_CLOCK	
_POSIX_NO_TRUNC	
_POSIX_REALTIME_SIGNALS	
_POSIX_SEMAPHORES	•
_POSIX_SHARED_MEMORY_OBJECTS	
_POSIX_SYNCHRONIZED_IO	
_POSIX_THREAD_ATTR_STACKADDR	
_POSIX_THREAD_ATTR_STACKSIZE	
_POSIX_THREAD_CPUTIME	
_POSIX_THREAD_PRIO_INHERIT	•
_POSIX_THREAD_PRIO_PROTECT	
_POSIX_THREAD_PRIORITY_SCHEDULING	•
_POSIX_THREAD_SPORADIC_SERVER	
_POSIX_TIMEOUTS	
_POSIX_TIMERS	
_POSIX2_C_DEV	
_POSIX2_SW_DEV	

Units of Functionality

This section describes the Units of Functionality (fixed sets of interfaces) which are implemented (partially or completely) in Zephyr. Please refer to the standard for a full description of each listed interface.

POSIX_THREADS_BASE The basic assumption in this profile is that the system consists of a single (implicit) process with multiple threads. Therefore, the standard requires all basic thread services, except those related to multiple processes.

Table 9: POSIX_THREADS_BASE

API	Supported
pthread_atfork()	
pthread_attr_destroy()	•
pthread_attr_getdetachstate()	•
pthread_attr_getschedparam()	•
pthread_attr_init()	•

continues on next page

Table 9 – continued from previous page

API	Supported
pthread_attr_setdetachstate()	•
pthread_attr_setschedparam()	•
pthread_cancel()	•
pthread_cleanup_pop()	
pthread_cleanup_push()	
pthread_cond_broadcast()	•
pthread_cond_destroy()	
pthread_cond_init()	•
pthread_cond_signal()	•
pthread_cond_timedwait()	•
pthread_cond_wait()	•
pthread_condattr_destroy()	
pthread_condattr_init()	
pthread_create()	•
pthread_detach()	•
pthread_equal()	
pthread_exit()	•
pthread_getspecific()	•
pthread_join()	•
pthread_key_create()	•
pthread_key_delete()	•
pthread_kill()	

continues on next page

Table 9 – continued from previous page

API	Supported
pthread_mutex_destroy()	•
pthread_mutex_init()	•
pthread_mutex_lock()	•
pthread_mutex_trylock()	•
pthread_mutex_unlock()	•
pthread_mutexattr_destroy()	
pthread_mutexattr_init()	
pthread_once()	•
pthread_self()	•
pthread_setcalcelstate()	
pthread_setcanceltype()	
pthread_setspecific()	•
pthread_sigmask()	
pthread_testcancel()	

XSI_THREAD_EXT The XSI_THREADS_EXT Unit of Functionality is required because it provides functions to control a thread's stack. This is considered useful for any real-time application.

This table lists service support status in Zephyr:

Table 10: XSI_THREAD_EXT

API	Supported
pthread_attr_getguardsize()	
pthread_attr_getstack()	•
pthread_attr_setguardsize()	
pthread_attr_setstack()	•
pthread_getconcurrency()	
pthread_setconcurrency()	

XSI_THREAD_MUTEX_EXT The XSI_THREAD_MUTEX_EXT Unit of Functionality is required because it has options for controlling the behavior of mutexes under erroneous application use.

This table lists service support status in Zephyr:

Table 11: XSI_THREAD_MUTEX_EXT

API	Supported
pthread_mutexattr_gettype()	•
pthread_mutexattr_settype()	•

POSIX_C_LANG_SUPPORT The POSIX_C_LANG_SUPPORT Unit of Functionality contains the general ISO C Library.

This is implemented as part of the minimal C library available in Zephyr.

Table 12: POSIX_C_LANG_SUPPORT

API	Supported
abs()	•
asctime()	
asctime_r()	
atof()	
atoi()	•
atol()	
atoll()	
bsearch()	•
calloc()	•
ctime()	
ctime_r()	
difftime()	
div()	
feclearexcept()	
fegetenv()	
fegetexceptflag()	
fegetround()	
feholdexcept()	
feraiseexcept()	
fesetenv()	
fesetexceptflag()	
fesetround()	
fetestexcept()	
feupdateenv()	
free()	•
gmtime()	•

continues on next page

Table 12 – continued from previous page

API	Supported
gmtime_r()	•
imaxabs()	
imaxdiv()	
isalnum()	•
isalpha()	•
isblank()	
iscntrl()	
isdigit()	•
isgraph()	•
islower()	
isprint()	•
ispunct()	
isspace()	•
isupper()	•
isxdigit()	•
labs()	•
ldiv()	
llabs()	•
lldiv()	
localeconv()	
localtime()	•
localtime_r()	
malloc()	•
memchr()	•

continues on next page

Table 12 – continued from previous page

API	Supported
memcmp()	•
memcpy()	•
memmove()	•
memset()	•
mktime()	•
qsort()	
rand()	•
rand_r()	
realloc()	•
setlocale()	
snprintf()	•
sprintf()	•
srand()	•
sscanf()	
strcat()	•
strchr()	•
strcmp()	•
strcoll()	
strcpy()	•
strcspn()	
strerror()	
strerror_r()	
strftime()	
strlen()	•

continues on next page

Table 12 – continued from previous page

API	Supported
strncat()	•
strncmp()	•
strncpy()	•
strpbrk()	
strrchr()	•
strspn()	
strstr()	•
strtod()	
strtof()	
strtoimax()	
strtok()	
strtok_r()	•
strtol()	•
strtold()	
strtoll()	
strtoul()	•
strtoull()	
strtoumax()	
strxfrm()	
time()	•
tolower()	•
toupper()	•
tzname()	
tzset()	
va_arg()	
va_copy()	
va_end()	
va_start()	
vsnprintf()	•

continues on next page

Table 12 – continued from previous page

API	Supported
vsprintf()	•
vsscanf()	

POSIX_SINGLE_PROCESS The POSIX_SINGLE_PROCESS Unit of Functionality contains services for single process applications.

Table 13: POSIX_SINGLE_PROCESS

API	Supported
confstr()	
environ	
errno	
getenv()	
setenv()	
sysconf()	
uname()	
unsetenv()	

POSIX_SIGNALS Signal services are a basic mechanism within POSIX-based systems and are required for error and event handling.

Table 14: POSIX_SIGNALS

API	Supported
abort()	•
alarm()	
kill()	
pause()	
raise()	
sigaction()	
igaddset()	
sigdelset()	
sigemptyset()	
sigfillset()	
igismember()	
signal()	
sigpending()	
sigprocmask()	
igsuspend()	
sigwait()	

POSIX_DEVICE_IO

Table 15: POSIX_DEVICE_IO

API	Supported
flockfile()	
ftrylockfile()	
funlockfile()	

continues on next page

Table 15 – continued from previous page

API	Supported
getc_unlocked()	
getchar_unlocked()	
putc_unlocked()	
putchar_unlocked()	
clearerr()	
close()	
fclose()	
fdopen()	
feof()	
ferror()	
fflush()	
fgetc()	
fgets()	
fileno()	
fopen()	
fprintf()	•
fputc()	•
fputs()	•
fread()	
freopen()	
fscanf()	
fwrite()	•
getc()	
getchar()	
gets()	
open()	•
perror()	
printf()	•
putc()	•
putchar()	
puts()	•
read()	•
scanf()	
setbuf()	
etvbuf()	
stderr	

continues on next page

Table 15 – continued from previous page

API	Supported
stdin	
stdout	
ungetc()	
vfprintf()	•
vfscanf()	
vprintf()	•
vscanf()	
write()	

8.16.2 CMSIS RTOS v1

Cortex-M Software Interface Standard (CMSIS) RTOS is a vendor-independent hardware abstraction layer for the ARM Cortex-M processor series and defines generic tool interfaces. Though it was originally defined for ARM Cortex-M microcontrollers alone, it could be easily extended to other microcontrollers making it generic. For more information on CMSIS RTOS v1, please refer http://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html

8.16.3 CMSIS RTOS v2

Cortex-M Software Interface Standard (CMSIS) RTOS is a vendor-independent hardware abstraction layer for the ARM Cortex-M processor series and defines generic tool interfaces. Though it was originally defined for ARM Cortex-M microcontrollers alone, it could be easily extended to other microcontrollers making it generic. For more information on CMSIS RTOS v2, please refer to the CMSIS-RTOS2 Documentation.

Features not supported in Zephyr implementation

Kernel `osKernelGetState`, `osKernelSuspend`, `osKernelResume`, `osKernelInitialize` and `osKernelStart` are not supported.

Mutex `osMutexPrioInherit` is supported by default and is not configurable, you cannot select/unselect this attribute.

`osMutexRecursive` is also supported by default. If this attribute is not set, an error is thrown when the same thread tries to acquire it the second time.

`osMutexRobust` is not supported in Zephyr.

Return values not supported in the Zephyr implementation

`osKernelUnlock`, `osKernelLock`, `osKernelRestoreLock` `osError` (Unspecified error) is not supported.

`osSemaphoreDelete` `osErrorResource` (the semaphore specified by parameter semaphore_id is in an invalid semaphore state) is not supported.

`osMutexDelete` `osErrorResource` (mutex specified by parameter mutex_id is in an invalid mutex state) is not supported.

`osTimerDelete` `osErrorResource` (the timer specified by parameter timer_id is in an invalid timer state) is not supported.

`osMessageQueueReset` `osErrorResource` (the message queue specified by parameter msgq_id is in an invalid message queue state) is not supported.

`osMessageQueueDelete` `osErrorResource` (the message queue specified by parameter msgq_id is in an invalid message queue state) is not supported.

`osMemoryPoolFree` `osErrorResource` (the memory pool specified by parameter mp_id is in an invalid memory pool state) is not supported.

`osMemoryPoolDelete` `osErrorResource` (the memory pool specified by parameter mp_id is in an invalid memory pool state) is not supported.

`osEventFlagsSet`, `osEventFlagsClear` `osFlagsErrorUnknown` (Unspecified error) and osFlagsErrorResource (Event flags object specified by parameter ef_id is not ready to be used) are not supported.

`osEventFlagsDelete` `osErrorParameter` (the value of the parameter ef_id is incorrect) is not supported.

`osThreadFlagsSet` `osFlagsErrorUnknown` (Unspecified error) and `osFlagsErrorResource` (Thread specified by parameter thread_id is not active to receive flags) are not supported.

`osThreadFlagsClear` `osFlagsErrorResource` (Running thread is not active to receive flags) is not supported.

`osDelayUntil` `osParameter` (the time cannot be handled) is not supported.

8.17 Porting

These pages document how to port Zephyr to new hardware.

8.17.1 Architecture Porting Guide

An architecture port is needed to enable Zephyr to run on an ISA (instruction set architecture) or an ABI (Application Binary Interface) that is not currently supported.

The following are examples of ISAs and ABIs that Zephyr supports:

- x86_32 ISA with System V ABI
- ARMv7-M ISA with Thumb2 instruction set and ARM Embedded ABI (aeabi)
- ARCv2 ISA

For information on Kconfig configuration, see *Setting Kconfig configuration values*. Architectures use a Kconfig configuration scheme similar to boards.

An architecture port can be divided in several parts; most are required and some are optional:

- **The early boot sequence**: each architecture has different steps it must take when the CPU comes out of reset (required).
- **Interrupt and exception handling**: each architecture handles asynchronous and unrequested events in a specific manner (required).
- **Thread context switching**: the Zephyr context switch is dependent on the ABI and each ISA has a different set of registers to save (required).
- **Thread creation and termination**: A thread's initial stack frame is ABI and architecture-dependent, and thread abortion possibly as well (required).
- **Device drivers**: most often, the system clock timer and the interrupt controller are tied to the architecture (some required, some optional).
- **Utility libraries**: some common kernel APIs rely on a architecture-specific implementation for performance reasons (required).

- **CPU idling/power management**: most architectures implement instructions for putting the CPU to sleep (partly optional, most likely very desired).
- **Fault management**: for implementing architecture-specific debug help and handling of fatal error in threads (partly optional).
- **Linker scripts and toolchains**: architecture-specific details will most likely be needed in the build system and when linking the image (required).

Early Boot Sequence

The goal of the early boot sequence is to take the system from the state it is after reset to a state where is can run C code and thus the common kernel initialization sequence. Most of the time, very few steps are needed, while some architectures require a bit more work to be performed.

Common steps for all architectures:

- Setup an initial stack.
- If running an XIP (eXecute-In-Place) kernel, copy initialized data
- from ROM to RAM.
- If not using an ELF loader, zero the BSS section.
- Jump to `_Cstart()`, the early kernel initialization
  - `_Cstart()` is responsible for context switching out of the fake context running at startup into the main thread.

Some examples of architecture-specific steps that have to be taken:

- If given control in real mode on x86_32, switch to 32-bit protected mode.
- Setup the segment registers on x86_32 to handle boot loaders that leave them in an unknown or broken state.
- Initialize a board-specific watchdog on Cortex-M3/4.
- Switch stacks from MSP to PSP on Cortex-M.
- Use a different approach than calling into _Swap() on Cortex-M to prevent race conditions.
- Setup FIRQ and regular IRQ handling on ARCv2.

Interrupt and Exception Handling

Each architecture defines interrupt and exception handling differently.

When a device wants to signal the processor that there is some work to be done on its behalf, it raises an interrupt. When a thread does an operation that is not handled by the serial flow of the software itself, it raises an exception. Both, interrupts and exceptions, pass control to a handler. The handler is known as an ISR (Interrupt Service Routine) in the case of interrupts. The handler performs the work required by the exception or the interrupt. For interrupts, that work is device-specific. For exceptions, it depends on the exception, but most often the core kernel itself is responsible for providing the handler.

The kernel has to perform some work in addition to the work the handler itself performs. For example:

- Prior to handing control to the handler:
  - Save the currently executing context.
  - Possibly getting out of power saving mode, which includes waking up devices.
  - Updating the kernel uptime if getting out of tickless idle mode.
- After getting control back from the handler:

  - Decide whether to perform a context switch.
  - When performing a context switch, restore the context being context switched in.

This work is conceptually the same across architectures, but the details are completely different:

- The registers to save and restore.
- The processor instructions to perform the work.
- The numbering of the exceptions.
- etc.

It thus needs an architecture-specific implementation, called the interrupt/exception stub.

Another issue is that the kernel defines the signature of ISRs as:

```
void (*isr)(void *parameter)
```

Architectures do not have a consistent or native way of handling parameters to an ISR. As such there are two commonly used methods for handling the parameter.

- Using some architecture defined mechanism, the parameter value is forced in the stub. This is commonly found in X86-based architectures.
- The parameters to the ISR are inserted and tracked via a separate table requiring the architecture to discover at runtime which interrupt is executing. A common interrupt handler demuxer is installed for all entries of the real interrupt vector table, which then fetches the device's ISR and parameter from the separate table. This approach is commonly used in the ARC and ARM architectures via the `CONFIG_GEN_ISR_TABLES` implementation. You can find examples of the stubs by looking at `_interrupt_enter()` in x86, `_IntExit()` in ARM, `_isr_wrapper()` in ARM, or the full implementation description for ARC in arch/arc/core/isr_wrapper.S.

Each architecture also has to implement primitives for interrupt control:

- locking interrupts: *irq_lock()*, *irq_unlock()*.
- registering interrupts: *IRQ_CONNECT()*.
- programming the priority if possible `irq_priority_set()`.
- enabling/disabling interrupts: *irq_enable()*, *irq_disable()*.

Note: *IRQ_CONNECT* is a macro that uses assembler and/or linker script tricks to connect interrupts at build time, saving boot time and text size.

The vector table should contain a handler for each interrupt and exception that can possibly occur. The handler can be as simple as a spinning loop. However, we strongly suggest that handlers at least print some debug information. The information helps figuring out what went wrong when hitting an exception that is a fault, like divide-by-zero or invalid memory access, or an interrupt that is not expected (*spurious interrupt*). See the ARM implementation in arch/arm/core/aarch32/cortex_m/fault.c for an example.

Thread Context Switching

Multi-threading is the basic purpose to have a kernel at all. Zephyr supports two types of threads: preemptible and cooperative.

Two crucial concepts when writing an architecture port are the following:

- Cooperative threads run at a higher priority than preemptible ones, and always preempt them.
- After handling an interrupt, if a cooperative thread was interrupted, the kernel always goes back to running that thread, since it is not preemptible.

A context switch can happen in several circumstances:

- When a thread executes a blocking operation, such as taking a semaphore that is currently unavailable.
- When a preemptible thread unblocks a thread of higher priority by releasing the object on which it was blocked.
- When an interrupt unblocks a thread of higher priority than the one currently executing, if the currently executing thread is preemptible.
- When a thread runs to completion.
- When a thread causes a fatal exception and is removed from the running threads. For example, referencing invalid memory,

Therefore, the context switching must thus be able to handle all these cases.

The kernel keeps the next thread to run in a "cache", and thus the context switching code only has to fetch from that cache to select which thread to run.

There are two types of context switches: *cooperative* and *preemptive*.

- A *cooperative* context switch happens when a thread willfully gives the control to another thread. There are two cases where this happens
  - When a thread explicitly yields.
  - When a thread tries to take an object that is currently unavailable and is willing to wait until the object becomes available.
- A *preemptive* context switch happens either because an ISR or a thread causes an operation that schedules a thread of higher priority than the one currently running, if the currently running thread is preemptible. An example of such an operation is releasing an object on which the thread of higher priority was waiting.

Note: Control is never taken from cooperative thread when one of them is the running thread.

A cooperative context switch is always done by having a thread call the `_Swap()` kernel internal symbol. When `_Swap` is called, the kernel logic knows that a context switch has to happen: `_Swap` does not check to see if a context switch must happen. Rather, `_Swap` decides what thread to context switch in. `_Swap` is called by the kernel logic when an object being operated on is unavailable, and some thread yielding/sleeping primitives.

Note: On x86 and Nios2, `_Swap` is generic enough and the architecture flexible enough that `_Swap` can be called when exiting an interrupt to provoke the context switch. This should not be taken as a rule, since neither the ARM Cortex-M or ARCv2 port do this.

Since `_Swap` is cooperative, the caller-saved registers from the ABI are already on the stack. There is no need to save them in the k_thread structure.

A context switch can also be performed preemptively. This happens upon exiting an ISR, in the kernel interrupt exit stub:

- `_interrupt_enter` on x86 after the handler is called.
- `_IntExit` on ARM.
- `_firq_exit` and `_rirq_exit` on ARCv2.

In this case, the context switch must only be invoked when the interrupted thread was preemptible, not when it was a cooperative one, and only when the current interrupt is not nested.

The kernel also has the concept of "locking the scheduler". This is a concept similar to locking the interrupts, but lighter-weight since interrupts can still occur. If a thread has locked the scheduler, is it temporarily non-preemptible.

So, the decision logic to invoke the context switch when exiting an interrupt is simple:

- If the interrupted thread is not preemptible, do not invoke it.
- Else, fetch the cached thread from the ready queue, and:
  - If the cached thread is not the current thread, invoke the context switch.
  - Else, do not invoke it.

This is simple, but crucial: if this is not implemented correctly, the kernel will not function as intended and will experience bizarre crashes, mostly due to stack corruption.

Note: If running a coop-only system, i.e. if `CONFIG_NUM_PREEMPT_PRIORITIES` is 0, no preemptive context switch ever happens. The interrupt code can be optimized to not take any scheduling decision when this is the case.

Thread Creation and Termination

To start a new thread, a stack frame must be constructed so that the context switch can pop it the same way it would pop one from a thread that had been context switched out. This is to be implemented in an architecture-specific `_new_thread` internal routine.

The thread entry point is also not to be called directly, i.e. it should not be set as the PC (program counter) for the new thread. Rather it must be wrapped in `_thread_entry`. This means that the PC in the stack frame shall be set to `_thread_entry`, and the thread entry point shall be passed as the first parameter to `_thread_entry`. The specifics of this depend on the ABI.

The need for an architecture-specific thread termination implementation depends on the architecture. There is a generic implementation, but it might not work for a given architecture.

One reason that has been encountered for having an architecture-specific implementation of thread termination is that aborting a thread might be different if aborting because of a graceful exit or because of an exception. This is the case for ARM Cortex-M, where the CPU has to be taken out of handler mode if the thread triggered a fatal exception, but not if the thread gracefully exits its entry point function.

This means implementing an architecture-specific version of *`k_thread_abort()`*, and setting the Kconfig option `CONFIG_ARCH_HAS_THREAD_ABORT` as needed for the architecture (e.g. see arch/arm/core/aarch32/cortex_m/Kconfig).

Thread Local Storage

To enable thread local storage on a new architecture:

1. Implement *`arch_tls_stack_setup()`* to setup the TLS storage area in stack. Refer to the toolchain documentation on how the storage area needs to be structured. Some helper functions can be used:
   - Function `z_tls_data_size()` returns the size needed for thread local variables (excluding any extra data required by toolchain and architecture).
   - Function `z_tls_copy()` prepares the TLS storage area for thread local variables. This only copies the variable themselves and does not do architecture and/or toolchain specific data.
2. In the context switching, grab the `tls` field inside the new thread's `struct k_thread` and put it into an appropriate register (or some other variable) for access to the TLS storage area. Refer to toolchain and architecture documentation on which registers to use.
3. In kconfig, add `select CONFIG_ARCH_HAS_THREAD_LOCAL_STORAGE` to kconfig related to the new architecture.
4. Run the `tests/kernel/threads/tls` to make sure the new code works.

Device Drivers

The kernel requires very few hardware devices to function. In theory, the only required device is the interrupt controller, since the kernel can run without a system clock. In practice, to get access to most, if not all, of the sanity check test suite, a system clock is needed as well. Since these two are usually tied to the architecture, they are part of the architecture port.

Interrupt Controllers There can be significant differences between the interrupt controllers and the interrupt concepts across architectures.

For example, x86 has the concept of an IDT and different interrupt controllers. The position of an interrupt in the IDT determines its priority.

On the other hand, the ARM Cortex-M has the NVIC (Nested Vectored Interrupt Controller) as part of the architecture definition. There is no need for an IDT-like table that is separate from the NVIC vector table. The position in the table has nothing to do with priority of an IRQ: priorities are programmable per-entry.

The ARCv2 has its interrupt unit as part of the architecture definition, which is somewhat similar to the NVIC. However, where ARC defines interrupts as having a one-to-one mapping between exception and interrupt numbers (i.e. exception 1 is IRQ1, and device IRQs start at 16), ARM has IRQ0 being equivalent to exception 16 (and weirdly enough, exception 1 can be seen as IRQ-15).

All these differences mean that very little, if anything, can be shared between architectures with regards to interrupt controllers.

System Clock x86 has APIC timers and the HPET as part of its architecture definition. ARM Cortex-M has the SYSTICK exception. Finally, ARCv2 has the timer0/1 device.

Kernel timeouts are handled in the context of the system clock timer driver's interrupt handler.

Console Over Serial Line There is one other device that is almost a requirement for an architecture port, since it is so useful for debugging. It is a simple polling, output-only, serial port driver on which to send the console (`printk`, `printf`) output.

It is not required, and a RAM console (`CONFIG_RAM_CONSOLE`) can be used to send all output to a circular buffer that can be read by a debugger instead.

Utility Libraries

The kernel depends on a few functions that can be implemented with very few instructions or in a lock-less manner in modern processors. Those are thus expected to be implemented as part of an architecture port.

- Atomic operators.
  - If instructions do exist for a given architecture, the implementation is configured using the `CONFIG_ATOMIC_OPERATIONS_ARCH` Kconfig option.
  - If instructions do not exist for a given architecture, a generic version that wraps *irq_lock()* or *irq_unlock()* around non-atomic operations exists. It is configured using the `CONFIG_ATOMIC_OPERATIONS_C` Kconfig option.
- Find-least-significant-bit-set and find-most-significant-bit-set.
  - If instructions do not exist for a given architecture, it is always possible to implement these functions as generic C functions.

It is possible to use compiler built-ins to implement these, but be careful they use the required compiler barriers.

CPU Idling/Power Management

The kernel provides support for CPU power management with two functions: *arch_cpu_idle()* and *arch_cpu_atomic_idle()*.

arch_cpu_idle() can be as simple as calling the power saving instruction for the architecture with interrupts unlocked, for example `hlt` on x86, `wfi` or `wfe` on ARM, `sleep` on ARC. This function can be called in a loop within a context that does not care if it get interrupted or not by an interrupt before going to sleep. There are basically two scenarios when it is correct to use this function:

- In a single-threaded system, in the only thread when the thread is not used for doing real work after initialization, i.e. it is sitting in a loop doing nothing for the duration of the application.
- In the idle thread.

arch_cpu_atomic_idle(), on the other hand, must be able to atomically re-enable interrupts and invoke the power saving instruction. It can thus be used in real application code, again in single-threaded systems.

Normally, idling the CPU should be left to the idle thread, but in some very special scenarios, these APIs can be used by applications.

Both functions must exist for a given architecture. However, the implementation can be simply the following steps, if desired:

1. unlock interrupts
2. NOP

However, a real implementation is strongly recommended.

Fault Management

In the event of an unhandled CPU exception, the architecture code must call into `z_fatal_error()`. This function dumps out architecture-agnostic information and makes a policy decision on what to do next by invoking `k_sys_fatal_error()`. This function can be overridden to implement application-specific policies that could include locking interrupts and spinning forever (the default implementation) or even powering off the system (if supported).

Toolchain and Linking

Toolchain support has to be added to the build system.

Some architecture-specific definitions are needed in include/toolchain/gcc.h. See what exists in that file for currently supported architectures.

Each architecture also needs its own linker script, even if most sections can be derived from the linker scripts of other architectures. Some sections might be specific to the new architecture, for example the SCB section on ARM and the IDT section on x86.

Memory Management

If the target platform enables paging and requires drivers to memory-map their I/O regions, `CONFIG_MMU` needs to be enabled and the *arch_mem_map()* API implemented.

Stack Objects

The presence of memory protection hardware affects how stack objects are created. All architecture ports must specify the required alignment of the stack pointer, which is some combination of CPU and

ABI requirements. This is defined in architecture headers with ARCH_STACK_PTR_ALIGN and is typically something small like 4, 8, or 16 bytes.

Two types of thread stacks exist:

- "kernel" stacks defined with *K_KERNEL_STACK_DEFINE()* and related APIs, which can host kernel threads running in supervisor mode or used as the stack for interrupt/exception handling. These have significantly relaxed alignment requirements and use less reserved data. No memory is reserved for prvilege elevation stacks.
- "thread" stacks which typically use more memory, but are capable of hosting thread running in user mode, as well as any use-cases for kernel stacks.

If CONFIG_USERSPACE is not enabled, "thread" and "kernel" stacks are equivalent.

Additional macros may be defined in the architecture layer to specify the alignment of the base of stack objects, any reserved data inside the stack object not used for the thread's stack buffer, and how to round up stack sizes to support user mode threads. In the absence of definitions some defaults are assumed:

- ARCH_KERNEL_STACK_RESERVED: default no reserved space
- ARCH_THREAD_STACK_RESERVED: default no reserved space
- ARCH_KERNEL_STACK_OBJ_ALIGN: default align to ARCH_STACK_PTR_ALIGN
- ARCH_THREAD_STACK_OBJ_ALIGN: default align to ARCH_STACK_PTR_ALIGN
- ARCH_THREAD_STACK_SIZE_ALIGN: default round up to ARCH_STACK_PTR_ALIGN

All stack creation macros are defined in terms of these.

Stack objects all have the following layout, with some regions potentially zero-sized depending on configuration. There are always two main parts: reserved memory at the beginning, and then the stack buffer itself. The bounds of some areas can only be determined at runtime in the context of its associated thread object. Other areas are entirely computable at build time.

Some architectures may need to carve-out reserved memory at runtime from the stack buffer, instead of unconditionally reserving it at build time, or to supplement an existing reserved area (as is the case with the ARM FPU). Such carve-outs will always be tracked in thread.stack_info.start. The region specified by thread.stack_info.start and thread.stack_info.size is always fully accessible by a user mode thread. thread.stack_info.delta denotes an offset which can be used to compute the initial stack pointer from the very end of the stack object, taking into account storage for TLS and ASLR random offsets.

```
+---------------------+ <- thread.stack_obj
| Reserved Memory     | } K_(THREAD|KERNEL)_STACK_RESERVED
+---------------------+
| Carved-out memory   |
|.....................| <- thread.stack_info.start
| Unused stack buffer |
|                     |
|.....................| <- thread's current stack pointer
| Used stack buffer   |
|                     |
|.....................| <- Initial stack pointer. Computable
| ASLR Random offset  |      with thread.stack_info.delta
+---------------------| <- thread.userspace_local_data
| Thread-local data   |
+---------------------+ <- thread.stack_info.start +
                            thread.stack_info.size
```

At present, Zephyr does not support stacks that grow upward.

No Memory Protection If no memory protection is in use, then the defaults are sufficient.

HW-based stack overflow detection This option uses hardware features to generate a fatal error if a thread in supervisor mode overflows its stack. This is useful for debugging, although for a couple reasons, you can't reliably make any assertions about the state of the system after this happens:

- The kernel could have been inside a critical section when the overflow occurs, leaving important global data structures in a corrupted state.
- For systems that implement stack protection using a guard memory region, it's possible to overshoot the guard and corrupt adjacent data structures before the hardware detects this situation.

To enable the `CONFIG_HW_STACK_PROTECTION` feature, the system must provide some kind of hardware-based stack overflow protection, and enable the `CONFIG_ARCH_HAS_STACK_PROTECTION` option.

Two forms of HW-based stack overflow detection are supported: dedicated CPU features for this purpose, or special read-only guard regions immediately preceding stack buffers.

`CONFIG_HW_STACK_PROTECTION` only catches stack overflows for supervisor threads. This is not required to catch stack overflow from user threads; `CONFIG_USERSPACE` is orthogonal.

This feature only detects supervisor mode stack overflows, including stack overflows when handling system calls. It doesn't guarantee that the kernel has not been corrupted. Any stack overflow in supervisor mode should be treated as a fatal error, with no assertions about the integrity of the overall system possible.

Stack overflows in user mode are recoverable (from the kernel's perspective) and require no special configuration; `CONFIG_HW_STACK_PROTECTION` only applies to catching overflows when the CPU is in sueprvisor mode.

CPU-based stack overflow detection If we are detecting stack overflows in supervisor mode via special CPU registers (like ARM's SPLIM), then the defaults are sufficient.

Guard-based stack overflow detection We are detecting supervisor mode stack overflows via special memory protection region located immediately before the stack buffer that generates an exception on write. Reserved memory will be used for the guard region.

`ARCH_KERNEL_STACK_RESERVED` should be defined to the minimum size of a memory protection region. On most ARM CPUs this is 32 bytes. `ARCH_KERNEL_STACK_OBJ_ALIGN` should also be set to the required alignment for this region.

MMU-based systems should not reserve RAM for the guard region and instead simply leave an non-present virtual page below every stack when it is mapped into the address space. The stack object will still need to be properly aligned and sized to page granularity.

```
+-----------------------------+ <- thread.stack_obj
| Guard reserved memory       | } K_KERNEL_STACK_RESERVED
+-----------------------------+
| Guard carve-out             |
|.............................| <- thread.stack_info.start
| Stack buffer                |
.                             .
```

Guard carve-outs for kernel stacks are uncommon and should be avoided if possible. They tend to be needed for two situations:

- The same stack may be re-purposed to host a user thread, in which case the guard is unnecessary and shouldn't be unconditionally reserved. This is the case when privilege elevation stacks are not inside the stack object.
- The required guard size is variable and depends on context. For example, some ARM CPUs have lazy floating point stacking during exceptions and may decrement the stack pointer by a large value without writing anything, completely overshooting a minimally-sized guard and corrupting

adjacent memory. Rather than unconditionally reserving a larger guard, the extra memory is carved out if the thread uses floating point.

User mode enabled Enabling user mode activates two new requirements:

- A separate fixed-sized privilege mode stack, specified by `CONFIG_PRIVILEGED_STACK_SIZE`, must be allocated that the user thread cannot access. It is used as the stack by the kernel when handling system calls. If stack guards are implemented, a stack guard region must be able to be placed before it, with support for carve-outs if necessary.
- The memory protection hardware must be able to program a region that exactly covers the thread's stack buffer, tracked in `thread.stack_info`. This implies that `ARCH_THREAD_STACK_SIZE_ADJUST()` will need to round up the requested stack size so that a region may cover it, and that `ARCH_THREAD_STACK_OBJ_ALIGN()` is also specified per the granularity of the memory protection hardware.

This becomes more complicated if the memory protection hardware requires that all memory regions be sized to a power of two, and aligned to their own size. This is common on older MPUs and is known with `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT`.

`thread.stack_info` always tracks the user-accessible part of the stack object, it must always be correct to program a memory protection region with user access using the range stored within.

Non power-of-two memory region requirements On systems without power-of-two region requirements, the reserved memory area for threads stacks defined by `K_THREAD_STACK_RESERVED` may be used to contain the privilege mode stack. The layout could be something like:

```
+------------------------------+ <- thread.stack_obj
| Other platform data          |
+------------------------------+
| Guard region (if enabled)    |
+------------------------------+
| Guard carve-out (if needed)  |
|..............................|
| Privilege elevation stack    |
+------------------------------| <- thread.stack_obj +
| Stack buffer                 |      K_THREAD_STACK_RESERVED =
.                              .      thread.stack_info.start
```

The guard region, and any carve-out (if needed) would be configured as a read-only region when the thread is created.

- If the thread is a supervisor thread, the privilege elevation region is just extra stack memory. An overflow will eventually crash into the guard region.
- If the thread is running in user mode, a memory protection region will be configured to allow user threads access to the stack buffer, but nothing before or after it. An overflow in user mode will crash into the privilege elevation stack, which the user thread has no access to. An overflow when handling a system call will crash into the guard region.

On an MMU system there should be no physical guards; the privilege mode stack will be mapped into kernel memory, and the stack buffer in the user part of memory, each with non-present virtual guard pages below them to catch runtime stack overflows.

Other platform data may be stored before the guard region, but this is highly discouraged if such data could be stored in `thread.arch` somewhere.

`ARCH_THREAD_STACK_RESERVED` will need to be defined to capture the size of the reserved region containing platform data, privilege elevation stacks, and guards. It must be appropriately sized such that an MPU region to grant user mode access to the stack buffer can be placed immediately after it.

Power-of-two memory region requirements Thread stack objects must be sized and aligned to the same power of two, without any reserved memory to allow efficient packing in memory. Thus, any guards in the thread stack must be completely carved out, and the privilege elevation stack must be allocated elsewhere.

`ARCH_THREAD_STACK_SIZE_ADJUST()` and `ARCH_THREAD_STACK_OBJ_ALIGN()` should both be defined to `Z_POW2_CEIL()`. `K_THREAD_STACK_RESERVED` must be 0.

For the privilege stacks, the `CONFIG_GEN_PRIV_STACKS` must be, enabled. For every thread stack found in the system, a corresponding fixed- size kernel stack used for handling system calls is generated. The address of the privilege stacks can be looked up quickly at runtime based on the thread stack address using `z_priv_stack_find()`. These stacks are laid out the same way as other kernel-only stacks.

```
+-----------------------------+ <- z_priv_stack_find(thread.stack_obj)
| Reserved memory             | } K_KERNEL_STACK_RESERVED
+-----------------------------+
| Guard carve-out (if needed) |
|.............................|
| Privilege elevation stack   |
|                             |
+-----------------------------+ <- z_priv_stack_find(thread.stack_obj) +
                                     K_KERNEL_STACK_RESERVED +
                                     CONFIG_PRIVILEGED_STACK_SIZE

+-----------------------------+ <- thread.stack_obj
| MPU guard carve-out         |
| (supervisor mode only)      |
|.............................| <- thread.stack_info.start
| Stack buffer                |
.                             .
```

The guard carve-out in the thread stack object is only used if the thread is running in supervisor mode. If the thread drops to user mode, there is no guard and the entire object is used as the stack buffer, with full access to the associated user mode thread and `thread.stack_info` updated appropriately.

User Mode Threads

To support user mode threads, several kernel-to-arch APIs need to be implemented, and the system must enable the `CONFIG_ARCH_HAS_USERSPACE` option. Please see the documentation for each of these functions for more details:

- *arch_buffer_validate()* to test whether the current thread has access permissions to a particular memory region
- *arch_user_mode_enter()* which will irreversibly drop a supervisor thread to user mode privileges. The stack must be wiped.
- *arch_syscall_oops()* which generates a kernel oops when system call parameters can't be validated, in such a way that the oops appears to be generated from where the system call was invoked in the user thread
- *arch_syscall_invoke0()* through *arch_syscall_invoke6()* invoke a system call with the appropriate number of arguments which must all be passed in during the privilege elevation via registers.
- *arch_is_user_context()* return nonzero if the CPU is currently running in user mode
- *arch_mem_domain_max_partitions_get()* which indicates the max number of regions for a memory domain. MMU systems have an unlimited amount, MPU systems have constraints on this.

Some architectures may need to update software memory management structures or modify hardware registers on another CPU when memory domain APIs are invoked. If so,

CONFIG_ARCH_MEM_DOMAIN_SYNCHRONOUS_API must be selected by the architecture and some additional APIs must be implemented. This is common on MMU systems and uncommon on MPU systems:

- arch_mem_domain_thread_add()
- arch_mem_domain_thread_remove()
- arch_mem_domain_partition_add()
- arch_mem_domain_partition_remove()

Please see the doxygen documentation of these APIs for details.

In addition to implementing these APIs, there are some other tasks as well:

- _new_thread() needs to spawn threads with *K_USER* in user mode
- On context switch, the outgoing thread's stack memory should be marked inaccessible to user mode by making the appropriate configuration changes in the memory management hardware.. The incoming thread's stack memory should likewise be marked as accessible. This ensures that threads can't mess with other thread stacks.
- On context switch, the system needs to switch between memory domains for the incoming and outgoing threads.
- Thread stack areas must include a kernel stack region. This should be inaccessible to user threads at all times. This stack will be used when system calls are made. This should be fixed size for all threads, and must be large enough to handle any system call.
- A software interrupt or some kind of privilege elevation mechanism needs to be established. This is closely tied to how the _arch_syscall_invoke macros are implemented. On system call, the appropriate handler function needs to be looked up in _k_syscall_table. Bad system call IDs should jump to the K_SYSCALL_BAD handler. Upon completion of the system call, care must be taken not to leak any register state back to user mode.

API Reference

Timing

group arch-timing

Unnamed Group

void arch_timing_init(void)

Initialize the timing subsystem.

Perform the necessary steps to initialize the timing subsystem.

See also:

timing_init()

void arch_timing_start(void)

Signal the start of the timing information gathering.

Signal to the timing subsystem that timing information will be gathered from this point forward.

See also:

timing_start()

void `arch_timing_stop`(void)

Signal the end of the timing information gathering.

Signal to the timing subsystem that timing information is no longer being gathered from this point forward.

See also:

timing_stop()

timing_t `arch_timing_counter_get`(void)

Return timing counter.

See also:

timing_counter_get()

Returns Timing counter.

uint64_t `arch_timing_cycles_get`(volatile timing_t *const start, volatile timing_t *const end)

Get number of cycles between `start` and `end`.

For some architectures or SoCs, the raw numbers from counter need to be scaled to obtain actual number of cycles.

See also:

timing_cycles_get()

Parameters

- `start` – Pointer to counter at start of a measured execution.
- `end` – Pointer to counter at stop of a measured execution.

Returns Number of cycles between start and end.

uint64_t `arch_timing_freq_get`(void)

Get frequency of counter used (in Hz).

See also:

timing_freq_get()

Returns Frequency of counter used for timing in Hz.

uint64_t `arch_timing_cycles_to_ns`(uint64_t cycles)

Convert number of `cycles` into nanoseconds.

See also:

timing_cycles_to_ns()

Parameters

- `cycles` – Number of cycles

Returns Converted time value

uint64_t `arch_timing_cycles_to_ns_avg`(uint64_t cycles, uint32_t count)

Convert number of `cycles` into nanoseconds with averaging.

See also:

timing_cycles_to_ns_avg()

Parameters

- `cycles` – Number of cycles
- `count` – Times of accumulated cycles to average over

Returns Converted time value

uint32_t `arch_timing_freq_get_mhz`(void)

Get frequency of counter used (in MHz).

See also:

timing_freq_get_mhz()

Returns Frequency of counter used for timing in MHz.

Functions

void `arch_busy_wait`(uint32_t usec_to_wait)

Architecture-specific implementation of busy-waiting

Parameters

- `usec_to_wait` – Wait period, in microseconds

static inline uint32_t `arch_k_cycle_get_32`(void)

Obtain the current cycle count, in units that are hardware-specific

See also:

k_cycle_get_32()

Threads

group `arch-threads`

Functions

void `arch_new_thread`(struct *k_thread* *thread, k_thread_stack_t *stack, char *stack_ptr,
k_thread_entry_t entry, void *p1, void *p2, void *p3)

Handle arch-specific logic for setting up new threads

The stack and arch-specific thread state variables must be set up such that a later attempt to switch to this thread will succeed and we will enter z_thread_entry with the requested thread and arguments as its parameters.

At some point in this function's implementation, z_setup_new_thread() must be called with the true bounds of the available stack buffer within the thread's stack object.

The provided stack pointer is guaranteed to be properly aligned with respect to the CPU and ABI requirements. There may be space reserved between the stack pointer and the bounds of the stack buffer for initial stack pointer randomization and thread-local storage.

Fields in thread->base will be initialized when this is called.

Parameters

- `thread` – Pointer to uninitialized struct *k_thread*
- `stack` – Pointer to the stack object
- `stack_ptr` – Aligned initial stack pointer
- `entry` – Thread entry function
- `p1` – 1st entry point parameter
- `p2` – 2nd entry point parameter
- `p3` – 3rd entry point parameter

static inline void `arch_switch`(void *switch_to, void **switched_from)

Cooperative context switch primitive

The action of *arch_switch()* should be to switch to a new context passed in the first argument, and save a pointer to the current context into the address passed in the second argument.

The actual type and interpretation of the switch handle is specified by the architecture. It is the same data structure stored in the "switch_handle" field of a newly-created thread in *arch_new_thread()*, and passed to the kernel as the "interrupted" argument to z_get_next_switch_handle().

Note that on SMP systems, the kernel uses the store through the second pointer as a synchronization point to detect when a thread context is completely saved (so another CPU can know when it is safe to switch). This store must be done AFTER all relevant state is saved, and must include whatever memory barriers or cache management code is required to be sure another CPU will see the result correctly.

The simplest implementation of *arch_switch()* is generally to push state onto the thread stack and use the resulting stack pointer as the switch handle. Some architectures may instead decide to use a pointer into the thread struct as the "switch handle" type. These can legally assume that the second argument to *arch_switch()* is the address of the switch_handle field of struct thread_base and can use an offset on this value to find other parts of the thread struct. For example a (C pseudocode) implementation of *arch_switch()* might look like:

void *arch_switch(void *switch_to, void **switched_from)* { struct *k_thread* *new = switch_to; struct *k_thread* *old = *CONTAINER_OF(switched_from, struct k_thread,switch_handle)*;

// save old context... *switched_from = old; // restore new context... }

Note that the kernel manages the switch_handle field for synchronization as described above. So it is not legal for architecture code to assume that it has any particular value at any other time. In particular it is not legal to read the field from the address passed in the second argument.

Parameters

- `switch_to` – Incoming thread's switch handle
- `switched_from` – Pointer to outgoing thread's switch handle storage location, which must be updated.

void `arch_switch_to_main_thread`(struct *k_thread* *main_thread, char *stack_ptr, k_thread_entry_t _main)

Custom logic for entering main thread context at early boot

Used by architectures where the typical trick of setting up a dummy thread in early boot context to "switch out" of isn't workable.

Parameters

- `main_thread` – main thread object
- `stack_ptr` – Initial stack pointer
- `_main` – Entry point for application main function.

int `arch_float_disable`(struct *k_thread* *thread)

Disable floating point context preservation.

The function is used to disable the preservation of floating point context information for a particular thread.

Note: For ARM architecture, disabling floating point preservation may only be requested for the current thread and cannot be requested in ISRs.

Return values

- `0` – On success.
- `-EINVAL` – If the floating point disabling could not be performed.
- `-ENOTSUP` – If the operation is not supported

int `arch_float_enable`(struct *k_thread* *thread, unsigned int options)

Enable floating point context preservation.

The function is used to enable the preservation of floating point context information for a particular thread. This API depends on each architecture implimentation. If the architecture does not support enabling, this API will always be failed.

The *options* parameter indicates which floating point register sets will be used by the specified thread. Currently it is used by x86 only.

Parameters

- `thread` – ID of thread.
- `options` – architecture dependent options

Return values

- `0` – On success.
- `-EINVAL` – If the floating point enabling could not be performed.
- `-ENOTSUP` – If the operation is not supported

group `arch-tls`

Functions

size_t `arch_tls_stack_setup`(struct *k_thread* *new_thread, char *stack_ptr)

Setup Architecture-specific TLS area in stack.

This sets up the stack area for thread local storage. The structure inside in area is architecture specific.

Parameters

- `new_thread` – New thread object
- `stack_ptr` – Stack pointer

Returns Number of bytes taken by the TLS area

Power Management

group `arch-pm`

Functions

FUNC_NORETURN void `arch_system_halt`(unsigned int reason)

Halt the system, optionally propagating a reason code

void `arch_cpu_idle`(void)

Power save idle routine.

This function will be called by the kernel idle loop or possibly within an implementation of z_pm_save_idle in the kernel when the '_pm_save_flag' variable is non-zero.

Architectures that do not implement power management instructions may immediately return, otherwise a power-saving instruction should be issued to wait for an interrupt.

See also:

k_cpu_idle()

Note: The function is expected to return after the interrupt that has caused the CPU to exit power-saving mode has been serviced, although this is not a firm requirement.

---

void `arch_cpu_atomic_idle`(unsigned int key)

Atomically re-enable interrupts and enter low power mode.

The requirements for *arch_cpu_atomic_idle()* are as follows:

a. Enabling interrupts and entering a low-power mode needs to be atomic, i.e. there should be no period of time where interrupts are enabled before the processor enters a low-power mode. See the comments in *k_lifo_get()*, for example, of the race condition that occurs if this requirement is not met.

b. After waking up from the low-power mode, the interrupt lockout state must be restored as indicated in the 'key' input parameter.

See also:

k_cpu_atomic_idle()

Parameters

- `key` – Lockout key returned by previous invocation of *arch_irq_lock()*

Symmetric Multi-Processing

group `arch-smp`

Typedefs

typedef FUNC_NORETURN void (*arch_cpustart_t)(void *data)

Per-cpu entry function

Param data context parameter, implementation specific

Functions

void arch_start_cpu(int cpu_num, k_thread_stack_t *stack, int sz, *arch_cpustart_t* fn, void *arg)

Start a numbered CPU on a MP-capable system.

This starts and initializes a specific CPU. The main thread on startup is running on CPU zero, other processors are numbered sequentially. On return from this function, the CPU is known to have begun operating and will enter the provided function. Its interrupts will be initialized but disabled such that *irq_unlock()* with the provided key will work to enable them.

Normally, in SMP mode this function will be called by the kernel initialization and should not be used as a user API. But it is defined here for special-purpose apps which want Zephyr running on one core and to use others for design-specific processing.

Parameters

- cpu_num – Integer number of the CPU
- stack – Stack memory for the CPU
- sz – Stack buffer size, in bytes
- fn – Function to begin running on the CPU.
- arg – Untyped argument to be passed to “fn”

bool arch_cpu_active(int cpu_num)

Return CPU power status.

Parameters

- cpu_num – Integer number of the CPU

static inline struct _cpu *arch_curr_cpu(void)

Return the CPU struct for the currently executing CPU

void arch_sched_ipi(void)

Broadcast an interrupt to all CPUs

This will invoke z_sched_ipi() on other CPUs in the system.

Interrupts

group arch-irq

Functions

static inline bool arch_is_in_isr(void)

Test if the current context is in interrupt context

XXX: This is inconsistently handled among arches wrt exception context See: #17656

Returns true if we are in interrupt context

static inline unsigned int `arch_irq_lock`(void)

Lock interrupts on the current CPU

See also:

irq_lock()

static inline void `arch_irq_unlock`(unsigned int key)

Unlock interrupts on the current CPU

See also:

irq_unlock()

static inline bool `arch_irq_unlocked`(unsigned int key)

Test if calling *arch_irq_unlock()* with this key would unlock irqs

Parameters

- `key` – value returned by *arch_irq_lock()*

Returns true if interrupts were unlocked prior to the *arch_irq_lock()* call that produced the key argument.

void `arch_irq_disable`(unsigned int irq)

Disable the specified interrupt line

See also:

irq_disable()

Note: : The behavior of interrupts that arrive after this call returns and before the corresponding call to *arch_irq_enable()* is undefined. The hardware is not required to latch and deliver such an interrupt, though on some architectures that may work. Other architectures will simply lose such an interrupt and never deliver it. Many drivers and subsystems are not tolerant of such dropped interrupts and it is the job of the application layer to ensure that behavior remains correct.

---

void `arch_irq_enable`(unsigned int irq)

Enable the specified interrupt line

See also:

irq_enable()

int `arch_irq_is_enabled`(unsigned int irq)

Test if an interrupt line is enabled

See also:

irq_is_enabled()

int `arch_irq_connect_dynamic`(unsigned int irq, unsigned int priority, void (*routine)(const void *parameter), const void *parameter, uint32_t flags)

Arch-specific hook to install a dynamic interrupt.

Parameters

- `irq` – IRQ line number
- `priority` – Interrupt priority
- `routine` – Interrupt service routine
- `parameter` – ISR parameter
- `flags` – Arch-specific IRQ configuration flag

Returns The vector assigned to this interrupt

Userspace

group `arch-userspace`

Functions

static inline uintptr_t `arch_syscall_invoke0`(uintptr_t call_id)

Invoke a system call with 0 arguments.

No general-purpose register state other than return value may be preserved when transitioning from supervisor mode back down to user mode for security reasons.

It is required that all arguments be stored in registers when elevating privileges from user to supervisor mode.

Processing of the syscall takes place on a separate kernel stack. Interrupts should be enabled when invoking the system call marshallers from the dispatch table. Thread preemption may occur when handling system calls.

Call ids are untrusted and must be bounds-checked, as the value is used to index the system call dispatch table, containing function pointers to the specific system call code.

Parameters

- `call_id` – System call ID

Returns Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke1`(uintptr_t arg1, uintptr_t call_id)

Invoke a system call with 1 argument.

See also:

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke2`(uintptr_t arg1, uintptr_t arg2, uintptr_t call_id)

Invoke a system call with 2 arguments.

See also:

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke3`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t call_id)

Invoke a system call with 3 arguments.

See also:

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke4`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t arg4, uintptr_t call_id)

Invoke a system call with 4 arguments.

See also:

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `arg4` – Fourth argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke5`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t arg4, uintptr_t arg5, uintptr_t call_id)

Invoke a system call with 5 arguments.

See also:

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `arg4` – Fourth argument to the system call.
- `arg5` – Fifth argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns Return value of the system call. Void system calls return 0 here.

static inline uintptr_t `arch_syscall_invoke6`(uintptr_t arg1, uintptr_t arg2, uintptr_t arg3, uintptr_t arg4, uintptr_t arg5, uintptr_t arg6, uintptr_t call_id)

Invoke a system call with 6 arguments.

See also:

arch_syscall_invoke0()

Parameters

- `arg1` – First argument to the system call.
- `arg2` – Second argument to the system call.
- `arg3` – Third argument to the system call.
- `arg4` – Fourth argument to the system call.
- `arg5` – Fifth argument to the system call.
- `arg6` – Sixth argument to the system call.
- `call_id` – System call ID, will be bounds-checked and used to reference kernel-side dispatch table

Returns Return value of the system call. Void system calls return 0 here.

static inline bool `arch_is_user_context`(void)

Indicate whether we are currently running in user mode

Returns true if the CPU is currently running with user permissions

int `arch_mem_domain_max_partitions_get`(void)

Get the maximum number of partitions for a memory domain.

Returns Max number of partitions, or -1 if there is no limit

int `arch_buffer_validate`(void *addr, size_t size, int write)

Check memory region permissions.

Given a memory region, return whether the current memory management hardware configuration would allow a user thread to read/write that region. Used by system calls to validate buffers coming in from userspace.

Notes: The function is guaranteed to never return validation success, if the entire buffer area is not user accessible.

The function is guaranteed to correctly validate the permissions of the supplied buffer, if the user access permissions of the entire buffer are enforced by a single, enabled memory management region.

In some architectures the validation will always return failure if the supplied memory buffer spans multiple enabled memory management regions (even if all such regions permit user access).

Warning: 0 size buffer has undefined behavior.

Parameters

- `addr` – start address of the buffer
- `size` – the size of the buffer
- `write` – If nonzero, additionally check if the area is writable. Otherwise, just check if the memory can be read.

Returns nonzero if the permissions don't match.

FUNC_NORETURN void `arch_user_mode_enter`(k_thread_entry_t user_entry, void *p1, void *p2, void *p3)

Perform a one-way transition from supervisor to kernel mode.

Implementations of this function must do the following:

- Reset the thread's stack pointer to a suitable initial value. We do not need any prior context since this is a one-way operation.
- Set up any kernel stack region for the CPU to use during privilege elevation
- Put the CPU in whatever its equivalent of user mode is
- Transfer execution to *arch_new_thread()* passing along all the supplied arguments, in user mode.

Parameters

- `user_entry` – Entry point to start executing as a user thread
- `p1` – 1st parameter to user thread
- `p2` – 2nd parameter to user thread
- `p3` – 3rd parameter to user thread

FUNC_NORETURN void `arch_syscall_oops`(void *ssf)

Induce a kernel oops that appears to come from a specific location.

Normally, k_oops() generates an exception that appears to come from the call site of the k_oops() itself.

However, when validating arguments to a system call, if there are problems we want the oops to appear to come from where the system call was invoked and not inside the validation function.

Parameters

- `ssf` – System call stack frame pointer. This gets passed as an argument to _k_syscall_handler_t functions and its contents are completely architecture specific.

size_t `arch_user_string_nlen`(const char *s, size_t maxsize, int *err)

Safely take the length of a potentially bad string.

This must not fault, instead the err parameter must have -1 written to it. This function otherwise should work exactly like libc strnlen(). On success *err should be set to 0.

Parameters

- `s` – String to measure
- `maxsize` – Max length of the string
- `err` – Error value to write

Returns Length of the string, not counting NULL byte, up to maxsize

static inline bool `arch_mem_coherent`(void *ptr)

Detect memory coherence type.

Required when ARCH_HAS_COHERENCE is true. This function returns true if the byte pointed to lies within an architecture-defined "coherence region" (typically implemented with uncached memory) and can safely be used in multiprocessor code without explicit flush or invalidate operations.

Note: The result is for only the single byte at the specified address, this API is not required to check region boundaries or to expect aligned pointers. The expectation is that the code above will have queried the appropriate address(es).

---

static inline void `arch_cohere_stacks`(struct *k_thread* *old_thread, void *old_switch_handle, struct *k_thread* *new_thread)

Ensure cache coherence prior to context switch.

Required when ARCH_HAS_COHERENCE is true. On cache-incoherent multiprocessor architectures, thread stacks are cached by default for performance reasons. They must therefore be flushed appropriately on context switch. The rules are:

a. The region containing live data in the old stack (generally the bytes between the current stack pointer and the top of the stack memory) must be flushed to underlying storage so a new CPU that runs the same thread sees the correct data. This must happen before the assignment of the switch_handle field in the thread struct which signals the completion of context switch.

b. Any data areas to be read from the new stack (generally the same as the live region when it was saved) should be invalidated (and NOT flushed!) in the data cache. This is because another CPU may have run or re-initialized the thread since this CPU suspended it, and any data present in cache will be stale.

- old_thread The old thread to be flushed before being allowed to run on other CPUs.
- old_switch_handle The switch handle to be stored into old_thread (it will not be valid until the cache is flushed so is not present yet). This will be NULL if inside z_swap() (because the *arch_switch()* has not saved it yet).
- new_thread The new thread to be invalidated before it runs locally.

Note: The kernel will call this function during interrupt exit when a new thread has been chosen to run, and also immediately before entering *arch_switch()* to effect a code-driven context switch. In the latter case, it is very likely that more data will be written to the old_thread stack region after this function returns but before the completion of the switch. Simply flushing naively here is not sufficient on many architectures and coordination with the *arch_switch()* implementation is likely required.

Memory Management

group arch-mmu

Functions

void arch_mem_map(void *virt, uintptr_t phys, size_t size, uint32_t flags)

Map physical memory into the virtual address space

This is a low-level interface to mapping pages into the address space. Behavior when providing unaligned addresses/sizes is undefined, these are assumed to be aligned to CONFIG_MMU_PAGE_SIZE.

The core kernel handles all management of the virtual address space; by the time we invoke this function, we know exactly where this mapping will be established. If the page tables already had mappings installed for the virtual memory region, these will be overwritten.

If the target architecture supports multiple page sizes, currently only the smallest page size will be used.

The memory range itself is never accessed by this operation.

This API must be safe to call in ISRs or exception handlers. Calls to this API are assumed to be serialized, and indeed all usage will originate from kernel/mm.c which handles virtual memory management.

Architectures are expected to pre-allocate page tables for the entire address space, as defined by CONFIG_KERNEL_VM_BASE and CONFIG_KERNEL_VM_SIZE. This operation should never require any kind of allocation for paging structures.

Validation of arguments should be done via assertions.

This API is part of infrastructure still under development and may change.

Parameters

- virt – Page-aligned Destination virtual address to map
- phys – Page-aligned Source physical address to map
- size – Page-aligned size of the mapped memory region in bytes
- flags – Caching, access and control flags, see K_MAP_* macros

void arch_mem_unmap(void *addr, size_t size)

Remove mappings for a provided virtual address range

This is a low-level interface for un-mapping pages from the address space. When this completes, the relevant page table entries will be updated as if no mapping was ever made for that memory range. No previous context needs to be preserved. This function must update mappings in all active page tables.

Behavior when providing unaligned addresses/sizes is undefined, these are assumed to be aligned to CONFIG_MMU_PAGE_SIZE.

Behavior when providing an address range that is not already mapped is undefined.

This function should never require memory allocations for paging structures, and it is not necessary to free any paging structures. Empty page tables due to all contained entries being un-mapped may remain in place.

Implementations must invalidate TLBs as necessary.

This API is part of infrastructure still under development and may change.

Parameters

- addr – Page-aligned base virtual address to un-map

- `size` – Page-aligned region size

int `arch_page_phys_get(void *virt, uintptr_t *phys)`

Get the mapped physical memory address from virtual address.

The function only needs to query the current set of page tables as the information it reports must be common to all of them if multiple page tables are in use. If multiple page tables are active it is unnecessary to iterate over all of them.

Unless otherwise specified, virtual pages have the same mappings across all page tables. Calling this function on data pages that are exceptions to this rule (such as the scratch page) is undefined behavior. Just check the currently installed page tables and return the information in that.

Parameters

- `virt` – Page-aligned virtual address
- `phys` – **[out]** Mapped physical address (can be NULL if only checking if virtual address is mapped)

Return values

- `0` – if mapping is found and valid
- `-EFAULT` – if virtual address is not mapped

Miscellaneous Architecture APIs

group `arch-misc`

Functions

int `arch_printk_char_out(int c)`

Early boot console output hook

Definition of this function is optional. If implemented, any invocation of printk() (or logging calls with CONFIG_LOG_MINIMAL which are backed by printk) will default to sending characters to this function. It is useful for early boot debugging before main serial or console drivers come up.

This can be overridden at runtime with __printk_hook_install().

The default __weak implementation of this does nothing.

Parameters

- `c` – Character to print

Returns The character printed

static inline void `arch_kernel_init(void)`

Architecture-specific kernel initialization hook

This function is invoked near the top of _Cstart, for additional architecture-specific setup before the rest of the kernel is brought up.

TODO: Deprecate, most arches are using a prep_c() function to do the same thing in a simpler way

static inline void `arch_nop(void)`

Do nothing and return. Yawn.

8.17.2 Board Porting Guide

To add Zephyr support for a new board, you at least need a *board directory* with various files in it. Files in the board directory inherit support for at least one SoC and all of its features. Therefore, Zephyr must support your SoC as well.

Boards, SoCs, etc.

Zephyr's hardware support hierarchy has these layers, from most to least specific:

- Board: a particular CPU instance and its peripherals in a concrete hardware specification
- SoC: the exact system on a chip the board's CPU is part of
- SoC series: a smaller group of tightly related SoCs
- SoC family: a wider group of SoCs with similar characteristics
- CPU core: a particular CPU in an architecture
- Architecture: an instruction set architecture

You can visualize the hierarchy like this:

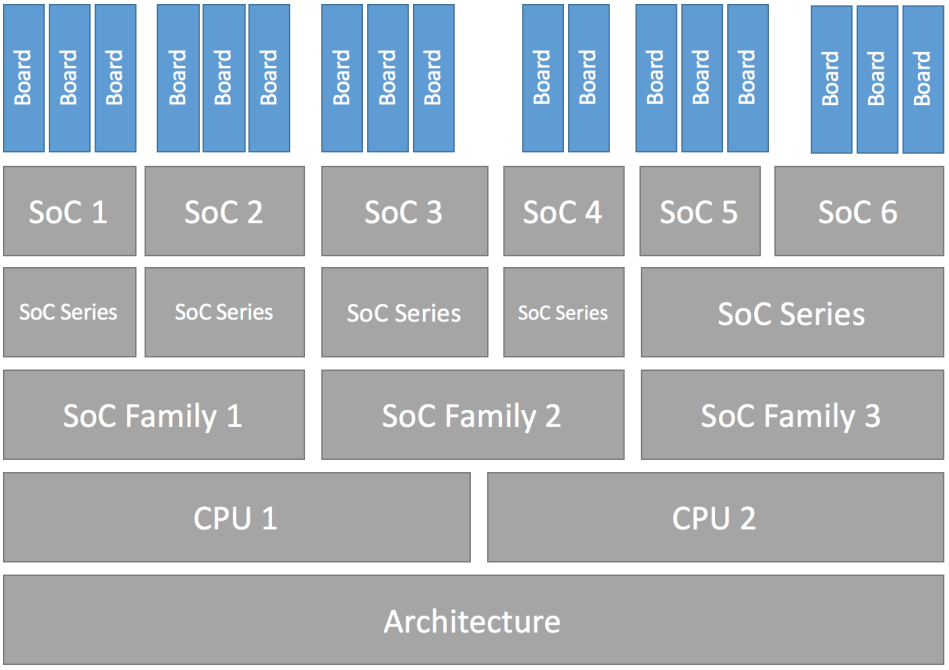


Fig. 12: Configuration Hierarchy

Here are some examples. Notice how the SoC series and family levels are not always used.

Board	SoC	SoC series	SoC family	CPU core	Architecture
nrf52dk_nrf52832	nRF52832	nRF52	Nordic nRF5	Arm Cortex-M4	Arm
frdm_k64f	MK64F12	Kinetis K6x	NXP Kinetis	Arm Cortex-M4	Arm
stm32h747i_disco	STM32H747XI	STM32H7	STMicro STM32	Arm Cortex-M7	Arm
rv32m1_vega_ri5cy	RV32M1	(Not used)	(Not used)	RI5CY	RISC-V

Make sure your SoC is supported

Start by making sure your SoC is supported by Zephyr. If it is, it's time to *Create your board directory*. If you don't know, try:

- checking boards for names that look relevant, and reading individual board documentation to find out for sure.
- asking your SoC vendor

If you need to add SoC, CPU core, or even architecture support, this is the wrong page, but here is some general advice.

Architecture See *Architecture Porting Guide*.

CPU Core CPU core support files go in `core` subdirectories under arch, e.g. arch/x86/core.

See *Set Up a Toolchain* for information about toolchains (compiler, linker, etc.) supported by Zephyr. If you need to support a new toolchain, *Build and Configuration Systems* is a good place to start learning about the build system. Please reach out to the community if you are looking for advice or want to collaborate on toolchain support.

SoC Zephyr SoC support files are in architecture-specific subdirectories of soc. They are generally grouped by SoC family.

When adding a new SoC family or series for a vendor that already has SoC support within Zephyr, please try to extract common functionality into shared files to avoid duplication. If there is no support for your vendor yet, you can add it in a new directory `zephyr/soc/<YOUR-ARCH>/<YOUR-SOC>`; please use self-explanatory directory names.

Create your board directory

Once you've found an existing board that uses your SoC, you can usually start by copy/pasting its board directory and changing its contents for your hardware.

You need to give your board a unique name. Run `west boards` for a list of names that are already taken, and pick something new. Let's say your board is called `plank` (please don't actually use that name).

Start by creating the board directory `zephyr/boards/<ARCH>/plank`, where `<ARCH>` is your SoC's architecture subdirectory. (You don't have to put your board directory in the zephyr repository, but it's the easiest way to get started. See *Custom Board, Devicetree and SOC Definitions* for documentation on moving your board directory to a separate repository once it's working.)

Your board directory should look like this:

```
boards/<ARCH>/plank
├── board.cmake
├── CMakeLists.txt
├── doc
│   ├── plank.png
│   └── index.rst
├── Kconfig.board
├── Kconfig.defconfig
├── plank_defconfig
├── plank.dts
└── plank.yaml
```

Replace `plank` with your board's name, of course.

The mandatory files are:

1. `plank.dts`: a hardware description in *devicetree* format. This declares your SoC, connectors, and any other hardware components such as LEDs, buttons, sensors, or communication peripherals (USB, BLE controller, etc).
2. `Kconfig.board`, `Kconfig.defconfig`, `plank_defconfig`: software configuration in *Configuration System (Kconfig)* formats. This provides default settings for software features and peripheral drivers.

The optional files are:

- `board.cmake`: used for *Flash and debug support*
- `CMakeLists.txt`: if you need to add additional source files to your build.

  One common use for this file is to add a `pinmux.c` file in your board directory to the build, which configures pin controllers at boot time. In that case, `CMakeLists.txt` usually looks like this:

```
if(CONFIG_PINMUX)
  zephyr_library()
  zephyr_library_sources(pinmux.c)
endif()
```

- `doc/index.rst`, `doc/plank.png`: documentation for and a picture of your board. You only need this if you're *Contributing your board* to Zephyr.
- `plank.yaml`: a YAML file with miscellaneous metadata used by the *Test Runner (Twister)*.

Write your devicetree

The devicetree file `boards/<ARCH>/plank/plank.dts` describes your board hardware in the Devicetree Source (DTS) format (as usual, change `plank` to your board's name). If you're new to devicetree, see *Introduction to devicetree*.

In general, `plank.dts` should look like this:

```
/dts-v1/;
#include <your_soc_vendor/your_soc.dtsi>

/ {
     model = "A human readable name";
     compatible = "yourcompany,plank";

     chosen {
             zephyr,console = &your_uart_console;
             zephyr,sram = &your_memory_node;
             /* other chosen settings  for your hardware */
     };

     /*
      * Your board-specific hardware: buttons, LEDs, sensors, etc.
      */

     leds {
             compatible = "gpio-leds";
             led0: led_0 {
                     gpios = < /* GPIO your LED is hooked up to */ >;
                     label = "LED 0";
             };
             /* ... other LEDs ... */
     };
```

(continues on next page)

(continued from previous page)

```
        buttons {
                compatible = "gpio-keys";
                /* ... your button definitions ... */
        };

        /* These aliases are provided for compatibility with samples */
        aliases {
                led0 = &led0; /* now you support the blinky sample! */
                /* other aliases go here */
        };
};

&some_peripheral_you_want_to_enable { /* like a GPIO or SPI controller */
        status = "okay";
};

&another_peripheral_you_want {
        status = "okay";
};
```

If you're in a hurry, simple hardware can usually be supported by copy/paste followed by trial and error. If you want to understand details, you will need to read the rest of the devicetree documentation and the devicetree specification.

Example: FRDM-K64F and Hexiwear K64 This section contains concrete examples related to writing your board's devicetree.

The FRDM-K64F and Hexiwear K64 board devicetrees are defined in frdm_k64fs.dts and hexiwear_k64.dts respectively. Both boards have NXP SoCs from the same Kinetis SoC family, the K6X.

Common devicetree definitions for K6X are stored in nxp_k6x.dtsi, which is included by both board `.dts` files. nxp_k6x.dtsi in turn includes armv7-m.dtsi, which has common definitions for Arm v7-M cores.

Since nxp_k6x.dtsi is meant to be generic across K6X-based boards, it leaves many devices disabled by default using `status` properties. For example, there is a CAN controller defined as follows (with unimportant parts skipped):

```
can0: can@40024000 {
        ...
        status = "disabled";
        ...
};
```

It is up to the board `.dts` or application overlay files to enable these devices as desired, by setting `status = "okay"`. The board `.dts` files are also responsible for any board-specific configuration of the device, such as adding nodes for on-board sensors, LEDs, buttons, etc.

For example, FRDM-K64 (but not Hexiwear K64) `.dts` enables the CAN controller and sets the bus speed:

```
&can0 {
        status = "okay";
        bus-speed = <125000>;
};
```

The `&can0 { ... };` syntax adds/overrides properties on the node with label `can0`, i.e. the `can@4002400` node defined in the `.dtsi` file.

Other examples of board-specific customization is pointing properties in `aliases` and `chosen` to the right nodes (see *Aliases and chosen nodes*), and making GPIO/pinmux assignments.

Write Kconfig files

Zephyr uses the Kconfig language to configure software features. Your board needs to provide some Kconfig settings before you can compile a Zephyr application for it.

Setting Kconfig configuration values is documented in detail in *Setting Kconfig configuration values*.

There are three mandatory Kconfig files in the board directory for a board named `plank`:

```
boards/<ARCH>/plank
├── Kconfig.board
├── Kconfig.defconfig
└── plank_defconfig
```

`Kconfig.board` Included by boards/Kconfig to include your board in the list of options.

This should at least contain a definition for a `BOARD_PLANK` option, which looks something like this:

```
config BOARD_PLANK
   bool "Plank board"
   depends on SOC_SERIES_YOUR_SOC_SERIES_HERE
   select SOC_PART_NUMBER_ABCDEFGH
```

`Kconfig.defconfig` Board-specific default values for Kconfig options.

The entire file should be inside an `if BOARD_PLANK` / `endif` pair of lines, like this:

```
if BOARD_PLANK

# Always set CONFIG_BOARD here. This isn't meant to be customized,
# but is set as a "default" due to Kconfig language restrictions.
config BOARD
   default "plank"

# Other options you want enabled by default go next. Examples:

config FOO
   default y

if NETWORKING
config SOC_ETHERNET_DRIVER
   default y
endif # NETWORKING

endif # BOARD_PLANK
```

`plank_defconfig` A Kconfig fragment that is merged as-is into the final build directory `.config` whenever an application is compiled for your board.

You should at least select your board's SOC and do any mandatory settings for your system clock, console, etc. The results are architecture-specific, but typically look something like this:

```
CONFIG_SOC_${VENDOR_XYZ3000}=y                    /* select your SoC */
CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC=120000000   /* set up your clock, etc */
CONFIG_SERIAL=y
```

`plank_x_y_z.conf` A Kconfig fragment that is merged as-is into the final build directory `.config` whenever an application is compiled for your board revision `x.y.z`.

Build, test, and fix

Now it's time to build and test the application(s) you want to run on your board until you're satisfied.

For example:

```
west build -b plank samples/hello_world
west flash
```

For `west flash` to work, see *Flash and debug support* below. You can also just flash `build/zephyr/zephyr.elf`, `zephyr.hex`, or `zephyr.bin` with any other tools you prefer.

General recommendations

For consistency and to make it easier for users to build generic applications that are not board specific for your board, please follow these guidelines while porting.

- Unless explicitly recommended otherwise by this section, leave peripherals and their drivers disabled by default.
- Configure and enable a system clock, along with a tick source.
- Provide pin and driver configuration that matches the board's valuable components such as sensors, buttons or LEDs, and communication interfaces such as USB, Ethernet connector, or Bluetooth/Wi-Fi chip.
- If your board uses a well-known connector standard (like Arduino, Mikrobus, Grove, or 96Boards connectors), add connector nodes to your DTS and configure pin muxes accordingly.
- Configure components that enable the use of these pins, such as configuring an SPI instance to use the usual Arduino SPI pins.
- If available, configure and enable a serial output for the console using the `zephyr,console` chosen node in the devicetree.
- If your board supports networking, configure a default interface.
- Enable all GPIO ports connected to peripherals or expansion connectors.
- If available, enable pinmux and interrupt controller drivers.
- It is recommended to enable the MPU by default, if there is support for it in hardware. For boards with limited memory resources it is acceptable to disable it. When the MPU is enabled, it is recommended to also enable hardware stack protection (CONFIG_HW_STACK_PROTECTION=y) and, thus, allow the kernel to detect stack overflows when the system is running in privileged mode.

Flash and debug support

Zephyr supports *Building, Flashing and Debugging* via west extension commands.

To add `west flash` and `west debug` support for your board, you need to create a `board.cmake` file in your board directory. This file's job is to configure a "runner" for your board. (There's nothing special you need to do to get `west build` support for your board.)

"Runners" are Zephyr-specific Python classes that wrap *flash and debug host tools* and integrate with west and the zephyr build system to support `west flash` and related commands. Each runner supports flashing, debugging, or both. You need to configure the arguments to these Python scripts in your `board.cmake` to support those commands like this example `board.cmake`:

```
board_runner_args(jlink "--device=nrf52" "--speed=4000")
board_runner_args(pyocd "--target=nrf52" "--frequency=4000000")
```

(continues on next page)

(continued from previous page)

```
include(${ZEPHYR_BASE}/boards/common/nrfjprog.board.cmake)
include(${ZEPHYR_BASE}/boards/common/jlink.board.cmake)
include(${ZEPHYR_BASE}/boards/common/pyocd.board.cmake)
```

This example configures the `nrfjprog`, `jlink`, and `pyocd` runners.

Warning: Runners usually have names which match the tools they wrap, so the `jlink` runner wraps Segger's J-Link tools, and so on. But the runner command line options like `--speed` etc. are specific to the Python scripts.

For more details:

- Run `west flash --context` to see a list of available runners which support flashing, and `west flash --context -r <RUNNER>` to view the specific options available for an individual runner.
- Run `west debug --context` and `west debug --context <RUNNER>` to get the same output for runners which support debugging.
- Run `west flash --help` and `west debug --help` for top-level options for flashing and debugging.
- See *Flash and debug runners* for Python APIs.
- Look for `board.cmake` files for other boards similar to your own for more examples.

To see what a `west flash` or `west debug` command is doing exactly, run it in verbose mode:

```
west --verbose flash
west --verbose debug
```

Verbose mode prints any host tool commands the runner uses.

The order of the `include()` calls in your `board.cmake` matters. The first `include` sets the default runner if it's not already set. For example, including `nrfjprog.board.cmake` first means that `nrjfprog` is the default flash runner for this board. Since `nrfjprog` does not support debugging, `jlink` is the default debug runner.

Multiple board revisions

See *Building for a board revision* for basics on this feature from the user perspective.

To create a new board revision for the `plank` board, create these additional files in the board folder:

```
boards/<ARCH>/plank
├── plank_<revision>.conf     # optional
├── plank_<revision>.overlay  # optional
└── revision.cmake
```

When the user builds for board `plank@<revision>`:

- The optional Kconfig settings specified in the file `plank_<revision>.conf` will be merged into the board's default Kconfig configuration.
- The optional devicetree overlay `plank_<revision>.overlay` will be added to the common `plank.dts` devicetree file
- The `revision.cmake` file controls how the Zephyr build system matches the `<board>@<revision>` string specified by the user when building an application for the board.

Currently, `<revision>` can be either a numeric `MAJOR.MINOR.PATCH` style revision like `1.5.0`, or single letter like `A`, `B`, etc. Zephyr provides a CMake board extension function, `board_check_revision()`, to make it easy to match either style from `revision.cmake`.

Valid board revisions may be specified as arguments to the `board_check_revision()` function, like:

```
board_check_revision(FORMAT MAJOR.MINOR.PATCH
                     VALID_REVISIONS 0.1.0 0.3.0 ...
)
```

Note: `VALID_REVISIONS` can be omitted if all valid revisions have specific Kconfig fragments, such as `<board>_0_1_0.conf`, `<board>_0_3_0.conf`. This allows you to just place Kconfig revision fragments in the board folder and not have to keep the corresponding `VALID_REVISIONS` in sync.

The following sections describe how to support these styles of revision numbers.

Numeric revisions Let's say you want to add support for revisions `0.5.0`, `1.0.0`, and `1.5.0` of the `plank` board with both Kconfig fragments and devicetree overlays. Create `revision.cmake` with `board_check_revision(FORMAT MAJOR.MINOR.PATCH)`, and create the following additional files in the board directory:

```
boards/<ARCH>/plank
├── plank_0_5_0.conf
├── plank_0_5_0.overlay
├── plank_1_0_0.conf
├── plank_1_0_0.overlay
├── plank_1_5_0.conf
├── plank_1_5_0.overlay
└── revision.cmake
```

Notice how the board files have changed periods (".") in the revision number to underscores ("_").

Fuzzy numeric revision matching To support "fuzzy" `MAJOR.MINOR.PATCH` revision matching for the `plank` board, use the following code in `revision.cmake`:

```
board_check_revision(FORMAT MAJOR.MINOR.PATCH)
```

If the user selects a revision between those available, the closest revision number that is not larger than the user's choice is used. For example, if the user builds for `plank@0.7.0`, the build system will target revision `0.5.0`.

The build system will print this at CMake configuration time:

```
-- Board: plank, Revision: 0.7.0 (Active: 0.5.0)
```

This allows you to only create revision configuration files for board revision numbers that introduce incompatible changes.

Any revision less than the minimum defined will be treated as an error.

You may use `0.0.0` as a minimum revision to build for by creating the file `plank_0_0_0.conf` in the board directory. This will be used for any revision lower than `0.5.0`, for example if the user builds for `plank@0.1.0`.

Exact numeric revision matching Alternatively, the `EXACT` keyword can be given to `board_check_revision()` in `revision.cmake` to allow exact matches only, like this:

```
board_check_revision(FORMAT MAJOR.MINOR.PATCH EXACT)
```

With this `revision.cmake`, building for `plank@0.7.0` in the above example will result in the following error message:

```
Board revision `0.7.0` not found.  Please specify a valid board revision.
```

Letter revision matching Let's say instead that you need to support revisions `A`, `B`, and `C` of the `plank` board. Create the following additional files in the board directory:

```
boards/<ARCH>/plank
├── plank_A.conf
├── plank_A.overlay
├── plank_B.conf
├── plank_B.overlay
├── plank_C.conf
├── plank_C.overlay
└── revision.cmake
```

And add the following to `revision.cmake`:

```
board_check_revision(FORMAT LETTER)
```

board_check_revision() details

```
board_check_revision(FORMAT <LETTER | MAJOR.MINOR.PATCH>
                     [EXACT]
                     [DEFAULT_REVISION <revision>]
                     [HIGHEST_REVISION <revision>]
                     [VALID_REVISIONS <revision> [<revision> ...]]
)
```

This function supports the following arguments:

- `FORMAT LETTER`: matches single letter revisions from `A` to `Z` only
- `FORMAT MAJOR.MINOR.PATCH`: matches exactly three digits. The command line allows for loose typing, that is `-DBOARD=<board>@1` and `-DBOARD=<board>@1.0` will be handled as `-DBOARD=<board>@1.0.0`. Kconfig fragment and devicetree overlay files must use full numbering to avoid ambiguity, so only `<board>_1_0_0.conf` and `<board>_1_0_0.overlay` are allowed.
- `EXACT`: if given, the revision is required to be an exact match. Otherwise, the closest matching revision not greater than the user's choice will be selected.
- `DEFAULT_REVISION <revision>`: if given, `<revision>` is the default revision to use when user has not selected a revision number. If not given, the build system prints an error when the user does not specify a board revision.
- `HIGHEST_REVISION`: if given, specifies the highest valid revision for a board. This can be used to ensure that a newer board cannot be used with an older Zephyr. For example, if the current board directory supports revisions 0.x.0-0.99.99 and 1.0.0-1.99.99, and it is expected that the implementation will not work with board revision 2.0.0, then giving `HIGHEST_REVISION 1.99.99` causes an error if the user builds using `<board>@2.0.0`.
- `VALID_REVISIONS`: if given, specifies a list of revisions that are valid for this board. If this argument is not given, then each Kconfig fragment of the form `<board>_<revision>.conf` in the board folder will be used as a valid revision for the board.

Custom revision.cmake files

Some boards may not use board revisions supported by `board_check_revision()`. To support revisions of any type, the file `revision.cmake` can implement custom revision matching without calling `board_check_revision()`.

To signal to the build system that it should use a different revision than the one specified by the user, `revision.cmake` can set the variable `ACTIVE_BOARD_REVISION` to the revision to use instead. The corresponding Kconfig files and devicetree overlays must be named `<board>_<ACTIVE_BOARD_REVISION>.conf` and `<board>_<ACTIVE_BOARD_REVISION>.overlay`.

For example, if the user builds for `plank@zero`, `revision.cmake` can set `ACTIVE_BOARD_REVISION` to `one` to use the files `plank_one.conf` and `plank_one.overlay`.

Contributing your board

If you want to contribute your board to Zephyr, first – thanks!

There are some extra things you'll need to do:

1. Make sure you've followed all the *General recommendations*. They are requirements for boards included with Zephyr.
2. Add documentation for your board using the template file doc/templates/board.tmpl. See *Documentation Generation* for information on how to build your documentation before submitting your pull request.
3. Prepare a pull request adding your board which follows the *Contribution Guidelines*.

8.17.3 Shields

Shields, also known as "add-on" or "daughter boards", attach to a board to extend its features and services for easier and modularized prototyping. In Zephyr, the shield feature provides Zephyr-formatted shield descriptions for easier compatibility with applications.

Shield porting and configuration

Shield configuration files are available in the board directory under /boards/shields:

```
boards/shields/<shield>
├── <shield>.overlay
├── Kconfig.shield
└── Kconfig.defconfig
```

These files provides shield configuration as follows:

- **<shield>.overlay**: This file provides a shield description in devicetree format that is merged with the board's *devicetree* before compilation.
- **Kconfig.shield**: This file defines shield Kconfig symbols that will be used for default shield configuration. To ease use with applications, the default shield configuration here should be consistent with those in the *Write your devicetree*.
- **Kconfig.defconfig**: This file defines the default shield configuration. It is made to be consistent with the *Write your devicetree*. Hence, shield configuration should be done by keeping in mind that features activation is application responsibility.

Board compatibility

Hardware shield-to-board compatibility depends on the use of well-known connectors used on popular boards (such as Arduino and 96boards). For software compatibility, boards must also provide a configuration matching their supported connectors.

This should be done at two different level:

- Pinmux: Connector pins should be correctly configured to match shield pins

- Devicetree: A board *devicetree* file, `BOARD.dts` should define a node alias for each connector interface. For example, for Arduino I2C:

```
#define arduino_i2c i2c1

aliases {
        arduino,i2c = &i2c1;
};
```

Note: With support of dtc v1.4.2, above will be replaced with the recently introduced overriding node element:

```
arduino_i2c:i2c1{};
```

Board specific shield configuration If modifications are needed to fit a shield to a particular board or board revision, you can override a shield description for a specific board by adding board or board revision overriding files to a shield, as follows:

```
boards/shields/<shield>
└── boards
    ├── <board>_<revision>.overlay
    ├── <board>.overlay
    ├── <board>.defconfig
    ├── <board>_<revision>.conf
    └── <board>.conf
```

Shield activation

Activate support for one or more shields by adding the matching -DSHIELD arg to CMake command

```
# From the root of the zephyr repository
west build -b None your_app -- -DSHIELD="x_nucleo_idb05a1 x_nucleo_iks01a1"
```

Alternatively, it could be set by default in a project's CMakeLists.txt:

```
set(SHIELD x_nucleo_iks01a1)
```

Shield variants

Some shields may support several variants or revisions. In that case, it is possible to provide multiple version of the shields description:

```
boards/shields/<shield>
├── <shield_v1>.overlay
├── <shield_v1>.defconfig
├── <shield_v2>.overlay
└── <shield_v2>.defconfig
```

In this case, a shield-particular revision name can be used:

```
# From the root of the zephyr repository
west build -b None your_app -- -DSHIELD=shield_v2
```

You can also provide a board-specific configuration to a specific shield revision:

```
boards/shields/<shield>
├── <shield_v1>.overlay
├── <shield_v1>.defconfig
├── <shield_v2>.overlay
├── <shield_v2>.defconfig
└── boards
    └── <shield_v2>
        ├── <board>.overlay
        └── <board>.defconfig
```

GPIO nexus nodes

GPIOs accessed by the shield peripherals must be identified using the shield GPIO abstraction, for example from the `arduino-header-r3` compatible. Boards that provide the header must map the header pins to SOC-specific pins. This is accomplished by including a nexus node that looks like the following into the board devicetree file:

```
arduino_header: connector {
        compatible = "arduino-header-r3";
        #gpio-cells = <2>;
        gpio-map-mask = <0xffffffff 0xffffffc0>;
        gpio-map-pass-thru = <0 0x3f>;
        gpio-map = <0 0 &gpioa 0 0>,    /* A0 */
                   <1 0 &gpioa 1 0>,    /* A1 */
                   <2 0 &gpioa 4 0>,    /* A2 */
                   <3 0 &gpiob 0 0>,    /* A3 */
                   <4 0 &gpioc 1 0>,    /* A4 */
                   <5 0 &gpioc 0 0>,    /* A5 */
                   <6 0 &gpioa 3 0>,    /* D0 */
                   <7 0 &gpioa 2 0>,    /* D1 */
                   <8 0 &gpioa 10 0>,   /* D2 */
                   <9 0 &gpiob 3 0>,    /* D3 */
                   <10 0 &gpiob 5 0>,   /* D4 */
                   <11 0 &gpiob 4 0>,   /* D5 */
                   <12 0 &gpiob 10 0>,  /* D6 */
                   <13 0 &gpioa 8 0>,   /* D7 */
                   <14 0 &gpioa 9 0>,   /* D8 */
                   <15 0 &gpioc 7 0>,   /* D9 */
                   <16 0 &gpiob 6 0>,   /* D10 */
                   <17 0 &gpioa 7 0>,   /* D11 */
                   <18 0 &gpioa 6 0>,   /* D12 */
                   <19 0 &gpioa 5 0>,   /* D13 */
                   <20 0 &gpiob 9 0>,   /* D14 */
                   <21 0 &gpiob 8 0>;   /* D15 */
};
```

This specifies how Arduino pin references like `<&arduino_header 11 0>` are converted to SOC gpio pin references like `<&gpiob 4 0>`.

In Zephyr GPIO specifiers generally have two parameters (indicated by `#gpio-cells = <2>`): the pin number and a set of flags. The low 6 bits of the flags correspond to features that can be configured in devicetree. In some cases it's necessary to use a non-zero flag value to tell the driver how a particular pin behaves, as with:

```
drdy-gpios = <&arduino_header 11 GPIO_ACTIVE_LOW>;
```

After preprocessing this becomes `<&arduino_header 11 1>`. Normally the presence of such a flag

would cause the map lookup to fail, because there is no map entry with a non-zero flags value. The `gpio-map-mask` property specifies that, for lookup, all bits of the pin and all but the low 6 bits of the flags are used to identify the specifier. Then the `gpio-map-pass-thru` specifies that the low 6 bits of the flags are copied over, so the SOC GPIO reference becomes `<&gpiob 4 1>` as intended.

See nexus node for more information about this capability.

8.18 Testing

8.18.1 Test Framework

The Zephyr Test Framework (Ztest) provides a simple testing framework intended to be used during development. It provides basic assertion macros and a generic test structure.

The framework can be used in two ways, either as a generic framework for integration testing, or for unit testing specific modules.

Quick start - Integration testing

A simple working base is located at samples/subsys/testsuite/integration. Just copy the files to `tests/` and edit them for your needs. The test will then be automatically built and run by the twister script. If you are testing the **bar** component of **foo**, you should copy the sample folder to `tests/foo/bar`. It can then be tested with:

```
./scripts/twister -s tests/foo/bar/test-identifier
```

In the example above `tests/foo/bar` signifies the path to the test and the `test-identifier` references a test defined in the `testcase.yaml` file.

To run all tests defined in a test project, run:

```
./scripts/twister -T tests/foo/bar/
```

The sample contains the following files:

CMakeLists.txt

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

cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(integration)

FILE(GLOB app_sources src/*.c)
target_sources(app PRIVATE ${app_sources})
```

testcase.yaml

```
tests:
  # section.subsection
  testing.ztest:
    build_only: true
    platform_allow: native_posix
    tags: testing
```

prj.conf

```
CONFIG_ZTEST=y
```

src/main.c (see *best practices*)

```
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <ztest.h>

/**
 * @brief Test Asserts
 *
 * This test verifies various assert macros provided by ztest.
 *
 */
static void test_assert(void)
{
        zassert_true(1, "1 was false");
        zassert_false(0, "0 was true");
        zassert_is_null(NULL, "NULL was not NULL");
        zassert_not_null("foo", "\"foo\" was NULL");
        zassert_equal(1, 1, "1 was not equal to 1");
        zassert_equal_ptr(NULL, NULL, "NULL was not equal to NULL");
}

void test_main(void)
{
        ztest_test_suite(framework_tests,
                ztest_unit_test(test_assert)
        );

        ztest_run_test_suite(framework_tests);
}
```

- *Listing Tests*
- *Skipping Tests*

A test case project may consist of multiple sub-tests or smaller tests that either can be testing functionality or APIs. Functions implementing a test should follow the guidelines below:

- Test cases function names should be prefix with **test_**
- Test cases should be documented using doxygen
- Test function names should be unique within the section or component being tested

An example can be seen below:

```
/**
 * @brief Test Asserts
 *
 * This test verifies the zassert_true macro.
 */
static void test_assert(void)
```

(continues on next page)

(continued from previous page)

```
{
        zassert_true(1, "1 was false");
}
```

The above test is then enabled as part of the testsuite using:

```
ztest_unit_test(test_assert)
```

Listing Tests Tests (test projects) in the Zephyr tree consist of many testcases that run as part of a project and test similar functionality, for example an API or a feature. The `twister` script can parse the testcases in all test projects or a subset of them, and can generate reports on a granular level, i.e. if cases have passed or failed or if they were blocked or skipped.

Twister parses the source files looking for test case names, so you can list all kernel test cases, for example, by entering:

```
twister --list-tests -T tests/kernel
```

Skipping Tests Special- or architecture-specific tests cannot run on all platforms and architectures, however we still want to count those and report them as being skipped. Because the test inventory and the list of tests is extracted from the code, adding conditionals inside the test suite is sub-optimal. Tests that need to be skipped for a certain platform or feature need to explicitly report a skip using *`ztest_test_skip()`*. If the test runs, it needs to report either a pass or fail. For example:

```
#ifdef CONFIG_TEST1
void test_test1(void)
{
        zassert_true(1, "true");
}
#else
void test_test1(void)
{
        ztest_test_skip();
}
#endif


void test_main(void)
{
        ztest_test_suite(common,
                         ztest_unit_test(test_test1),
                         ztest_unit_test(test_test2)
                         );
        ztest_run_test_suite(common);
}
```

Quick start - Unit testing

Ztest can be used for unit testing. This means that rather than including the entire Zephyr OS for testing a single function, you can focus the testing efforts into the specific module in question. This will speed up testing since only the module will have to be compiled in, and the tested functions will be called directly.

Since you won't be including basic kernel data structures that most code depends on, you have to provide function stubs in the test. Ztest provides some helpers for mocking functions, as demonstrated below.

In a unit test, mock objects can simulate the behavior of complex real objects and are used to decide whether a test failed or passed by verifying whether an interaction with an object occurred, and if required, to assert the order of that interaction.

Best practices for declaring the test suite *twister* and other validation tools need to obtain the list of subcases that a Zephyr *ztest* test image will expose.

Rationale

This all is for the purpose of traceability. It's not enough to have only a semaphore test project. We also need to show that we have testpoints for all APIs and functionality, and we trace back to documentation of the API, and functional requirements.

The idea is that test reports show results for every sub-testcase as passed, failed, blocked, or skipped. Reporting on only the high-level test project level, particularly when tests do too many things, is too vague.

Here is a generic template for a test showing the expected use of `ztest_test_suite()`:

```
#include <ztest.h>

extern void test_sometest1(void);
extern void test_sometest2(void);
#ifndef CONFIG_WHATEVER              /* Conditionally skip test_sometest3 */
void test_sometest3(void)
{
     ztest_test_skip();
}
#else
extern void test_sometest3(void);
#endif
extern void test_sometest4(void);
...

void test_main(void)
{
     ztest_test_suite(common,
                      ztest_unit_test(test_sometest1),
                      ztest_unit_test(test_sometest2),
                      ztest_unit_test(test_sometest3),
                      ztest_unit_test(test_sometest4)
                );
     ztest_run_test_suite(common);
}
```

For *twister* to parse source files and create a list of subcases, the declarations of `ztest_test_suite()` must follow a few rules:

- one declaration per line
- conditional execution by using `ztest_test_skip()`

What to avoid:

- packing multiple testcases in one source file

```
void test_main(void)
{
#ifdef TEST_feature1
```

(continues on next page)

(continued from previous page)

```
        ztest_test_suite(feature1,
                         ztest_unit_test(test_1a),
                         ztest_unit_test(test_1b),
                         ztest_unit_test(test_1c)
                         );
        ztest_run_test_suite(feature1);
#endif

#ifdef TEST_feature2
        ztest_test_suite(feature2,
                         ztest_unit_test(test_2a),
                         ztest_unit_test(test_2b)
                         );
        ztest_run_test_suite(feature2);
#endif
}
```

- Do not use `#if`

```
        ztest_test_suite(common,
                         ztest_unit_test(test_sometest1),
                         ztest_unit_test(test_sometest2),
#ifdef CONFIG_WHATEVER
                         ztest_unit_test(test_sometest3),
#endif
                         ztest_unit_test(test_sometest4),
        ...
```

- Do not add comments on lines with a call to `ztest_unit_test()`:

```
ztest_test_suite(common,
                 ztest_unit_test(test_sometest1),
                 ztest_unit_test(test_sometest2) /* will fail */,
/* will fail! */ ztest_unit_test(test_sometest3),
                 ztest_unit_test(test_sometest4),
...
```

- Do not define multiple definitions of unit / user unit test case per line

```
ztest_test_suite(common,
                 ztest_unit_test(test_sometest1), ztest_unit_test(test_
↪sometest2),
                 ztest_unit_test(test_sometest3),
                 ztest_unit_test(test_sometest4),
...
```

Other questions:

- Why not pre-scan with CPP and then parse? or post scan the ELF file?

  If C pre-processing or building fails because of any issue, then we won't be able to tell the subcases.

- Why not declare them in the YAML testcase description?

  A separate testcase description file would be harder to maintain than just keeping the information in the test source files themselves – only one file to update when changes are made eliminates duplication.

API reference

Running tests

group `ztest_test`

This module eases the testing process by providing helpful macros and other testing structures.

Defines

`ztest_unit_test_setup_teardown(fn, setup, teardown)`

Define a test with setup and teardown functions.

This should be called as an argument to ztest_test_suite. The test will be run in the following order: *setup*, *fn*, *teardown*.

Parameters

- `fn` – Main test function
- `setup` – Setup function
- `teardown` – Teardown function

`ztest_user_unit_test_setup_teardown(fn, setup, teardown)`

Define a user mode test with setup and teardown functions.

This should be called as an argument to ztest_test_suite. The test will be run in the following order: *setup*, *fn*, *teardown*. ALL test functions will be run in user mode, and only if CONFIG_USERSPACE is enabled, otherwise this is the same as *ztest_unit_test_setup_teardown()*.

Parameters

- `fn` – Main test function
- `setup` – Setup function
- `teardown` – Teardown function

`ztest_unit_test(fn)`

Define a test function.

This should be called as an argument to ztest_test_suite.

Parameters

- `fn` – Test function

`ztest_user_unit_test(fn)`

Define a test function that should run as a user thread.

This should be called as an argument to ztest_test_suite. If CONFIG_USERSPACE is not enabled, this is functionally identical to *ztest_unit_test()*.

Parameters

- `fn` – Test function

`ztest_1cpu_unit_test(fn)`

Define a SMP-unsafe test function.

As *ztest_unit_test()*, but ensures all test code runs on only one CPU when in SMP.

Parameters

- `fn` – Test function

ztest_1cpu_user_unit_test(fn)

Define a SMP-unsafe test function that should run as a user thread.

As *ztest_user_unit_test()*, but ensures all test code runs on only one CPU when in SMP.

Parameters

- fn – Test function

ZTEST_DMEM

ZTEST_BMEM

ZTEST_SECTION

ztest_test_suite(suite, ...)

Define a test suite.

This function should be called in the following fashion:

```
ztest_test_suite(test_suite_name,
        ztest_unit_test(test_function),
        ztest_unit_test(test_other_function)
);

ztest_run_test_suite(test_suite_name);
```

Parameters

- suite – Name of the testing suite

ztest_run_test_suite(suite)

Run the specified test suite.

Parameters

- suite – Test suite to run.

Functions

void ztest_test_fail(void)

Fail the currently running test.

This is the function called from failed assertions and the like. You probably don't need to call it yourself.

void ztest_test_pass(void)

Pass the currently running test.

Normally a test passes just by returning without an assertion failure. However, if the success case for your test involves a fatal fault, you can call this function from k_sys_fatal_error_handler to indicate that the test passed before aborting the thread.

void ztest_test_skip(void)

Skip the current test.

static inline void unit_test_noop(void)

Do nothing, successfully.

Unit test / setup function / teardown function that does nothing, successfully. Can be used as a parameter to *ztest_unit_test_setup_teardown()*.

Variables

struct *k_mem_partition* ztest_mem_partition

Assertions These macros will instantly fail the test if the related assertion fails. When an assertion fails, it will print the current file, line and function, alongside a reason for the failure and an optional message. If the config option:CONFIG_ZTEST_ASSERT_VERBOSE is 0, the assertions will only print the file and line numbers, reducing the binary size of the test.

Example output for a failed macro from zassert_equal(buf->ref, 2, "Invalid refcount"):

```
Assertion failed at main.c:62: test_get_single_buffer: Invalid refcount (buf->ref not␣
↪equal to 2)
Aborted at unit test function
```

group ztest_assert

This module provides assertions when using Ztest.

Defines

zassert(cond, default_msg, msg, ...)

Fail the test, if *cond* is false.

You probably don't need to call this macro directly. You should instead use zassert_{condition} macros below.

Note that when CONFIG_MULTITHREADING=n macro returns from the function. It is then expected that in that case ztest asserts will be used only in the context of the test function.

Parameters

- cond – Condition to check
- msg – Optional, can be NULL. Message to print if *cond* is false.
- default_msg – Message to print if *cond* is false

zassert_unreachable(msg, ...)

Assert that this function call won't be reached.

Parameters

- msg – Optional message to print if the assertion fails

zassert_true(cond, msg, ...)

Assert that *cond* is true.

Parameters

- cond – Condition to check
- msg – Optional message to print if the assertion fails

zassert_false(cond, msg, ...)

Assert that *cond* is false.

Parameters

- cond – Condition to check
- msg – Optional message to print if the assertion fails

`zassert_ok`(cond, msg, ...)

Assert that *cond* is 0 (success)

Parameters

- `cond` – Condition to check
- `msg` – Optional message to print if the assertion fails

`zassert_is_null`(ptr, msg, ...)

Assert that *ptr* is NULL.

Parameters

- `ptr` – Pointer to compare
- `msg` – Optional message to print if the assertion fails

`zassert_not_null`(ptr, msg, ...)

Assert that *ptr* is not NULL.

Parameters

- `ptr` – Pointer to compare
- `msg` – Optional message to print if the assertion fails

`zassert_equal`(a, b, msg, ...)

Assert that *a* equals *b*.

a and *b* won't be converted and will be compared directly.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `msg` – Optional message to print if the assertion fails

`zassert_not_equal`(a, b, msg, ...)

Assert that *a* does not equal *b*.

a and *b* won't be converted and will be compared directly.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `msg` – Optional message to print if the assertion fails

`zassert_equal_ptr`(a, b, msg, ...)

Assert that *a* equals *b*.

a and *b* will be converted to `void *` before comparing.

Parameters

- `a` – Value to compare
- `b` – Value to compare
- `msg` – Optional message to print if the assertion fails

`zassert_within`(a, b, d, msg, ...)

Assert that *a* is within *b* with delta *d*.

Parameters

- `a` – Value to compare

- `b` – Value to compare
- `d` – Delta
- `msg` – Optional message to print if the assertion fails

`zassert_mem_equal(...)`

Assert that 2 memory buffers have the same contents.

This macro calls the final memory comparison assertion macro. Using double expansion allows providing some arguments by macros that would expand to more than one values (ANSI-C99 defines that all the macro arguments have to be expanded before macro call).

Parameters

- `...` – Arguments, see *zassert_mem_equal__* for real arguments accepted.

`zassert_mem_equal__(buf, exp, size, msg, ...)`

Internal assert that 2 memory buffers have the same contents.

Note: This is internal macro, to be used as a second expansion. See *zassert_mem_equal*.

Parameters

- `buf` – Buffer to compare
- `exp` – Buffer with expected contents
- `size` – Size of buffers
- `msg` – Optional message to print if the assertion fails

Mocking These functions allow abstracting callbacks and related functions and controlling them from specific tests. You can enable the mocking framework by setting `CONFIG_ZTEST_MOCKING` to "y" in the configuration file of the test. The amount of concurrent return values and expected parameters is limited by `CONFIG_ZTEST_PARAMETER_COUNT`.

Here is an example for configuring the function `expect_two_parameters` to expect the values `a=2` and `b=3`, and telling `returns_int` to return `5`:

```
#include <ztest.h>

static void expect_two_parameters(int a, int b)
{
        ztest_check_expected_value(a);
        ztest_check_expected_value(b);
}

static void parameter_tests(void)
{
        ztest_expect_value(expect_two_parameters, a, 2);
        ztest_expect_value(expect_two_parameters, b, 3);
        expect_two_parameters(2, 3);
}

static int returns_int(void)
{
        return ztest_get_return_value();
}

static void return_value_tests(void)
```

(continues on next page)

(continued from previous page)

```
{
        ztest_returns_value(returns_int, 5);
        zassert_equal(returns_int(), 5, NULL);
}

void test_main(void)
{
        ztest_test_suite(mock_framework_tests,
                ztest_unit_test(parameter_test),
                ztest_unit_test(return_value_test)
        );

        ztest_run_test_suite(mock_framework_tests);
}
```

group `ztest_mock`

This module provides simple mocking functions for unit testing. These need CONFIG_ZTEST_MOCKING=y.

Defines

`ztest_expect_value`(func, param, value)

Tell function *func* to expect the value *value* for *param*.

When using *ztest_check_expected_value()*, tell that the value of *param* should be *value*. The value will internally be stored as an `uintptr_t`.

Parameters

- `func` – Function in question
- `param` – Parameter for which the value should be set
- `value` – Value for *param*

`ztest_check_expected_value`(param)

If *param* doesn't match the value set by *ztest_expect_value()*, fail the test.

This will first check that does *param* have a value to be expected, and then checks whether the value of the parameter is equal to the expected value. If either of these checks fail, the current test will fail. This must be called from the called function.

Parameters

- `param` – Parameter to check

`ztest_expect_data`(func, param, data)

Tell function *func* to expect the data *data* for *param*.

When using *ztest_check_expected_data()*, the data pointed to by *param* should be same *data* in this function. Only data pointer is stored by this function, so it must still be valid when ztest_check_expected_data is called.

Parameters

- `func` – Function in question
- `param` – Parameter for which the data should be set
- `data` – pointer for the data for parameter *param*

`ztest_check_expected_data`(param, length)

If data pointed by *param* don't match the data set by *ztest_expect_data()*, fail the test.

This will first check that *param* is expected to be null or non-null and then check whether the data pointed by parameter is equal to expected data. If either of these checks fail, the current test will fail. This must be called from the called function.

Parameters

- `param` – Parameter to check
- `length` – Length of the data to compare

`ztest_return_data`(func, param, data)

Tell function *func* to return the data *data* for *param*.

When using *ztest_return_data()*, the data pointed to by *param* should be same *data* in this function. Only data pointer is stored by this function, so it must still be valid when ztest_copy_return_data is called.

Parameters

- `func` – Function in question
- `param` – Parameter for which the data should be set
- `data` – pointer for the data for parameter *param*

`ztest_copy_return_data`(param, length)

Copy the data set by ztest_return_data to the memory pointed by *param*.

This will first check that *param* is not null and then copy the data. This must be called from the called function.

Parameters

- `param` – Parameter to return data for
- `length` – Length of the data to return

`ztest_returns_value`(func, value)

Tell *func* that it should return *value*.

Parameters

- `func` – Function that should return *value*
- `value` – Value to return from *func*

`ztest_get_return_value`()

Get the return value for current function.

The return value must have been set previously with *ztest_returns_value()*. If no return value exists, the current test will fail.

Returns The value the current function should return

`ztest_get_return_value_ptr`()

Get the return value as a pointer for current function.

The return value must have been set previously with *ztest_returns_value()*. If no return value exists, the current test will fail.

Returns The value the current function should return as a `void *`

Customizing Test Output

The way output is presented when running tests can be customized. An example can be found in tests/ztest/custom_output.

Customization is enabled by setting `CONFIG_ZTEST_TC_UTIL_USER_OVERRIDE` to "y" and adding a file `tc_util_user_override.h` with your overrides.

Add the line `zephyr_include_directories(my_folder)` to your project's `CMakeLists.txt` to let Zephyr find your header file during builds.

See the file subsys/testsuite/include/tc_util.h to see which macros and/or defines can be overridden. These will be surrounded by blocks such as:

```
#ifndef SOMETHING
#define SOMETHING <default implementation>
#endif /* SOMETHING */
```

8.18.2 Test Runner (Twister)

This script scans for the set of unit test applications in the git repository and attempts to execute them. By default, it tries to build each test case on boards marked as default in the board definition file.

The default options will build the majority of the tests on a defined set of boards and will run in an emulated environment if available for the architecture or configuration being tested.

In normal use, twister runs a limited set of kernel tests (inside an emulator). Because of its limited test execution coverage, twister cannot guarantee local changes will succeed in the full build environment, but it does sufficient testing by building samples and tests for different boards and different configurations to help keep the complete code tree buildable.

When using (at least) one `-v` option, twister's console output shows for every test how the test is run (qemu, native_posix, etc.) or whether the binary was just built. There are a few reasons why twister only builds a test and doesn't run it:

- The test is marked as `build_only: true` in its `.yaml` configuration file.
- The test configuration has defined a `harness` but you don't have it or haven't set it up.
- The target device is not connected and not available for flashing
- You or some higher level automation invoked twister with `--build-only`.

These also affect the outputs of `--testcase-report` and `--detailed-report`, see their respective `--help` sections.

To run the script in the local tree, follow the steps below:

```
$ source zephyr-env.sh
$ ./scripts/twister
```

If you have a system with a large number of cores, you can build and run all possible tests using the following options:

```
$ ./scripts/twister --all --enable-slow
```

This will build for all available boards and run all applicable tests in a simulated (for example QEMU) environment.

The list of command line options supported by twister can be viewed using:

```
$ ./scripts/twister --help
```

Board Configuration

To build tests for a specific board and to execute some of the tests on real hardware or in an emulation environment such as QEMU a board configuration file is required which is generic enough to be used for other tasks that require a board inventory with details about the board and its configuration that is only available during build time otherwise.

The board metadata file is located in the board directory and is structured using the YAML markup language. The example below shows a board with a data required for best test coverage for this specific board:

```
identifier: frdm_k64f
name: NXP FRDM-K64F
type: mcu
arch: arm
toolchain:
  - zephyr
  - gnuarmemb
  - xtools
supported:
  - arduino_gpio
  - arduino_i2c
  - netif:eth
  - adc
  - i2c
  - nvs
  - spi
  - gpio
  - usb_device
  - watchdog
  - can
  - pwm
testing:
  default: true
```

identifier: A string that matches how the board is defined in the build system. This same string is used when building, for example when calling `west build` or `cmake`:

```
# with west
west build -b reel_board
# with cmake
cmake -DBOARD=reel_board ..
```

name: The actual name of the board as it appears in marketing material.

type: Type of the board or configuration, currently we support 2 types: mcu, qemu

arch: Architecture of the board

toolchain: The list of supported toolchains that can build this board. This should match one of the values used for 'ZEPHYR_TOOLCHAIN_VARIANT' when building on the command line

ram: Available RAM on the board (specified in KB). This is used to match testcase requirements. If not specified we default to 128KB.

flash: Available FLASH on the board (specified in KB). This is used to match testcase requirements. If not specified we default to 512KB.

supported: A list of features this board supports. This can be specified as a single word feature or as a variant of a feature class. For example:

```
supported:
  - pci
```

This indicates the board does support PCI. You can make a testcase build or run only on such boards, or:

```
supported:
  - netif:eth
  - sensor:bmi16
```

A testcase can both depend on 'eth' to only test ethernet or on 'netif' to run on any board with a networking interface.

testing: testing relating keywords to provide best coverage for the features of this board.

default: [True|False]: This is a default board, it will tested with the highest priority and is covered when invoking the simplified twister without any additional arguments.

ignore_tags: Do not attempt to build (and therefore run) tests marked with this list of tags.

only_tags: Only execute tests with this list of tags on a specific platform.

Test Cases

Test cases are detected by the presence of a 'testcase.yaml' or a 'sample.yaml' files in the application's project directory. This file may contain one or more entries in the test section each identifying a test scenario.

The name of each testcase needs to be unique in the context of the overall testsuite and has to follow basic rules:

1. The format of the test identifier shall be a string without any spaces or special characters (allowed characters: alphanumric and [_=]) consisting of multiple sections delimited with a dot (.).
2. Each test identifier shall start with a section followed by a subsection separated by a dot. For example, a test that covers semaphores in the kernel shall start with `kernel.sempahore`.
3. All test identifiers within a testcase.yaml file need to be unique. For example a testcase.yaml file covering semaphores in the kernel can have:
   - `kernel.semaphore`: For general semaphore tests
   - `kernel.semaphore.stress`: Stress testng semaphores in the kernel.
4. Depending on the nature of the test, an identifier can consist of at least two sections:
   - Ztest tests: The individual testcases in the ztest testsuite will be concatenated to identifier in the testcase.yaml file generating unique identifiers for every testcase in the suite.
   - Standalone tests and samples: This type of test should at least have 3 sections in the test identifier in the testcase.yaml (or sample.yaml) file. The last section of the name shall signify the test itself.

Test cases are written using the YAML syntax and share the same structure as samples. The following is an example test with a few options that are explained in this document.

```
tests:
  bluetooth.gatt:
    build_only: true
    platform_allow: qemu_cortex_m3 qemu_x86
    tags: bluetooth
  bluetooth.gatt.br:
    build_only: true
```

(continues on next page)

(continued from previous page)

```
    extra_args: CONF_FILE="prj_br.conf"
    filter: not CONFIG_DEBUG
    platform_exclude: up_squared
    platform_allow: qemu_cortex_m3 qemu_x86
    tags: bluetooth
```

A sample with tests will have the same structure with additional information related to the sample and what is being demonstrated:

```
sample:
  name: hello world
  description: Hello World sample, the simplest Zephyr application
tests:
  sample.basic.hello_world:
    build_only: true
    tags: tests
    min_ram: 16
  sample.basic.hello_world.singlethread:
    build_only: true
    extra_args: CONF_FILE=prj_single.conf
    filter: not CONFIG_BT
    tags: tests
    min_ram: 16
```

The full canonical name for each test case is:

```
<path to test case>/<test entry>
```

Each test block in the testcase meta data can define the following key/value pairs:

tags: <list of tags> (required) A set of string tags for the testcase. Usually pertains to functional domains but can be anything. Command line invocations of this script can filter the set of tests to run based on tag.

skip: <True|False> (default False) skip testcase unconditionally. This can be used for broken tests.

slow: <True|False> (default False) Don't run this test case unless –enable-slow was passed in on the command line. Intended for time-consuming test cases that are only run under certain circumstances, like daily builds. These test cases are still compiled.

extra_args: <list of extra arguments> Extra arguments to pass to Make when building or running the test case.

extra_configs: <list of extra configurations> Extra configuration options to be merged with a master prj.conf when building or running the test case. For example:

```
common:
  tags: drivers adc
tests:
  test:
    depends_on: adc
  test_async:
    extra_configs:
      - CONFIG_ADC_ASYNC=y
```

build_only: <True|False> (default False) If true, don't try to run the test even if the selected platform supports it.

build_on_all: <True|False> (default False) If true, attempt to build test on all available platforms.

depends_on: <list of features> A board or platform can announce what features it supports, this option will enable the test only those platforms that provide this feature.

min_ram: <integer> minimum amount of RAM in KB needed for this test to build and run. This is compared with information provided by the board metadata.

min_flash: <integer> minimum amount of ROM in KB needed for this test to build and run. This is compared with information provided by the board metadata.

timeout: <number of seconds> Length of time to run test in QEMU before automatically killing it. Default to 60 seconds.

arch_allow: <list of arches, such as x86, arm, arc> Set of architectures that this test case should only be run for.

arch_exclude: <list of arches, such as x86, arm, arc> Set of architectures that this test case should not run on.

platform_allow: <list of platforms> Set of platforms that this test case should only be run for. Do not use this option to limit testing or building in CI due to time or resource constraints, this option should only be used if the test or sample can only be run on the allowed platform and nothing else.

integration_platforms: <YML list of platforms/boards> This option limits the scope to the listed platforms when twister is invoked with the –integration option. Use this instead of platform_allow if the goal is to limit scope due to timing or resource constraints.

platform_exclude: <list of platforms> Set of platforms that this test case should not run on.

extra_sections: <list of extra binary sections> When computing sizes, twister will report errors if it finds extra, unexpected sections in the Zephyr binary unless they are named here. They will not be included in the size calculation.

harness: <string> A harness string needed to run the tests successfully. This can be as simple as a loopback wiring or a complete hardware test setup for sensor and IO testing. Usually pertains to external dependency domains but can be anything such as console, sensor, net, keyboard, Bluetooth or pytest.

harness_config: <harness configuration options> Extra harness configuration options to be used to select a board and/or for handling generic Console with regex matching. Config can announce what features it supports. This option will enable the test to run on only those platforms that fulfill this external dependency.

The following options are currently supported:

type: <one_line|multi_line> (required) Depends on the regex string to be matched

record: <recording options>

regex: <expression> (required) Any string that the particular test case prints to record test results.

regex: <expression> (required) Any string that the particular test case prints to confirm test runs as expected.

ordered: <True|False> (default False) Check the regular expression strings in orderly or randomly fashion

repeat: <integer> Number of times to validate the repeated regex expression

fixture: <expression> Specify a test case dependency on an external device(e.g., sensor), and identify setups that fulfill this dependency. It depends on specific test setup and board selection logic to pick the particular board(s) out of multiple boards that fulfill the dependency in an automation setup based on “fixture” keyword. Some sample fixture names are i2c_hts221, i2c_bme280, i2c_FRAM, ble_fw and gpio_loop.

Only one fixture can be defined per testcase.

pytest_root: <pytest dirctory> (default pytest) Specify a pytest directory which need to excute when test case begin to running, default pytest directory name is pytest, after pytest finished, twister will check if this case pass or fail according the pytest report.

The following is an example yaml file with a few harness_config options.

```
sample:
  name: HTS221 Temperature and Humidity Monitor
common:
  tags: sensor
  harness: console
  harness_config:
    type: multi_line
    ordered: false
    regex:
      - "Temperature:(.*)C"
      - "Relative Humidity:(.*)%"
    fixture: i2c_hts221
tests:
  test:
    tags: sensors
    depends_on: i2c
```

The following is an example yaml file with pytest harness_config options, default pytest_root name "pytest" will be used if pytest_root not specified. please refer the example in samples/subsys/testsuite/pytest/.

```
tests:
  pytest.example:
    harness: pytest
    harness_config:
      pytest_root: [pytest directory name]
```

filter: <expression> Filter whether the testcase should be run by evaluating an expression against an environment containing the following values:

```
{ ARCH : <architecture>,
  PLATFORM : <platform>,
  <all CONFIG_* key/value pairs in the test's generated defconfig>,
  *<env>: any environment variable available
}
```

The grammar for the expression language is as follows:

expression ::= expression "and" expression

expression "or" expression
"not" expression
"(" expression ")"
symbol "==" constant
symbol "!=" constant
symbol "<" number
symbol ">" number
symbol ">=" number
symbol "<=" number
symbol "in" list
symbol ":" string
symbol

list ::= “[” list_contents “]”

list_contents ::= constant

list_contents “,” constant

constant ::= number

string

For the case where expression ::= symbol, it evaluates to true if the symbol is defined to a non-empty string.

Operator precedence, starting from lowest to highest:

or (left associative) and (left associative) not (right associative) all comparison operators (non-associative)

arch_allow, arch_exclude, platform_allow, platform_exclude are all syntactic sugar for these expressions. For instance

arch_exclude = x86 arc

Is the same as:

filter = not ARCH in [“x86”, “arc”]

The ‘:’ operator compiles the string argument as a regular expression, and then returns a true value only if the symbol’s value in the environment matches. For example, if CONFIG_SOC=”stm32f107xc” then

filter = CONFIG_SOC : “stm.*”

Would match it.

The set of test cases that actually run depends on directives in the testcase filed and options passed in on the command line. If there is any confusion, running with -v or examining the discard report (`twister_discard.csv`) can help show why particular test cases were skipped.

Metrics (such as pass/fail state and binary size) for the last code release are stored in scripts/release/twister_last_release.csv. To update this, pass the –all –release options.

To load arguments from a file, write ‘+’ before the file name, e.g., +file_name. File content must be one or more valid arguments separated by line break instead of white spaces.

Most everyday users will run with no arguments.

Running in Integration Mode

This mode is used in continuous integration (CI) and other automated environments used to give developers fast feedback on changes. The mode can be activated using the –integration option of twister and narrows down the scope of builds and tests if applicable to platforms defined under the integration keyword in the testcase definition file (testcase.yaml and sample.yaml).

Running Tests on Hardware

Beside being able to run tests in QEMU and other simulated environments, twister supports running most of the tests on real devices and produces reports for each run with detailed FAIL/PASS results.

Executing tests on a single device To use this feature on a single connected device, run twister with the following new options:

```
scripts/twister --device-testing --device-serial /dev/ttyACM0 -p \
frdm_k64f  -T tests/kernel
```

The `--device-serial` option denotes the serial device the board is connected to. This needs to be accessible by the user running twister. You can run this on only one board at a time, specified using the `--platform` option.

Executing tests on multiple devices To build and execute tests on multiple devices connected to the host PC, a hardware map needs to be created with all connected devices and their details such as the serial device and their IDs if available. Run the following command to produce the hardware map:

```
./scripts/twister --generate-hardware-map map.yml
```

The generated hardware map file (map.yml) will have the list of connected devices, for example:

```
- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: unknown
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
- connected: true
  id: 000683759358
  platform: unknown
  product: J-Link
  runner: unknown
  serial: /dev/cu.usbmodem0006837593581
```

Any options marked as 'unknown' need to be changed and set with the correct values, in the above example both the platform names and the runners need to be replaced with the correct values corresponding to the connected hardware. In this example we are using a reel_board and an nrf52840dk_nrf52840:

```
- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: reel_board
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
- connected: true
  id: 000683759358
  platform: nrf52840dk_nrf52840
  product: J-Link
  runner: nrfjprog
  serial: /dev/cu.usbmodem0006837593581
```

If the map file already exists, then new entries are added and existing entries will be updated. This way you can use one single master hardware map and update it for every run to get the correct serial devices and status of the devices.

With the hardware map ready, you can run any tests by pointing to the map file:

```
./scripts/twister --device-testing --hardware-map map.yml -T samples/hello_world/
```

The above command will result in twister building tests for the platforms defined in the hardware map and subsequently flashing and running the tests on those platforms.

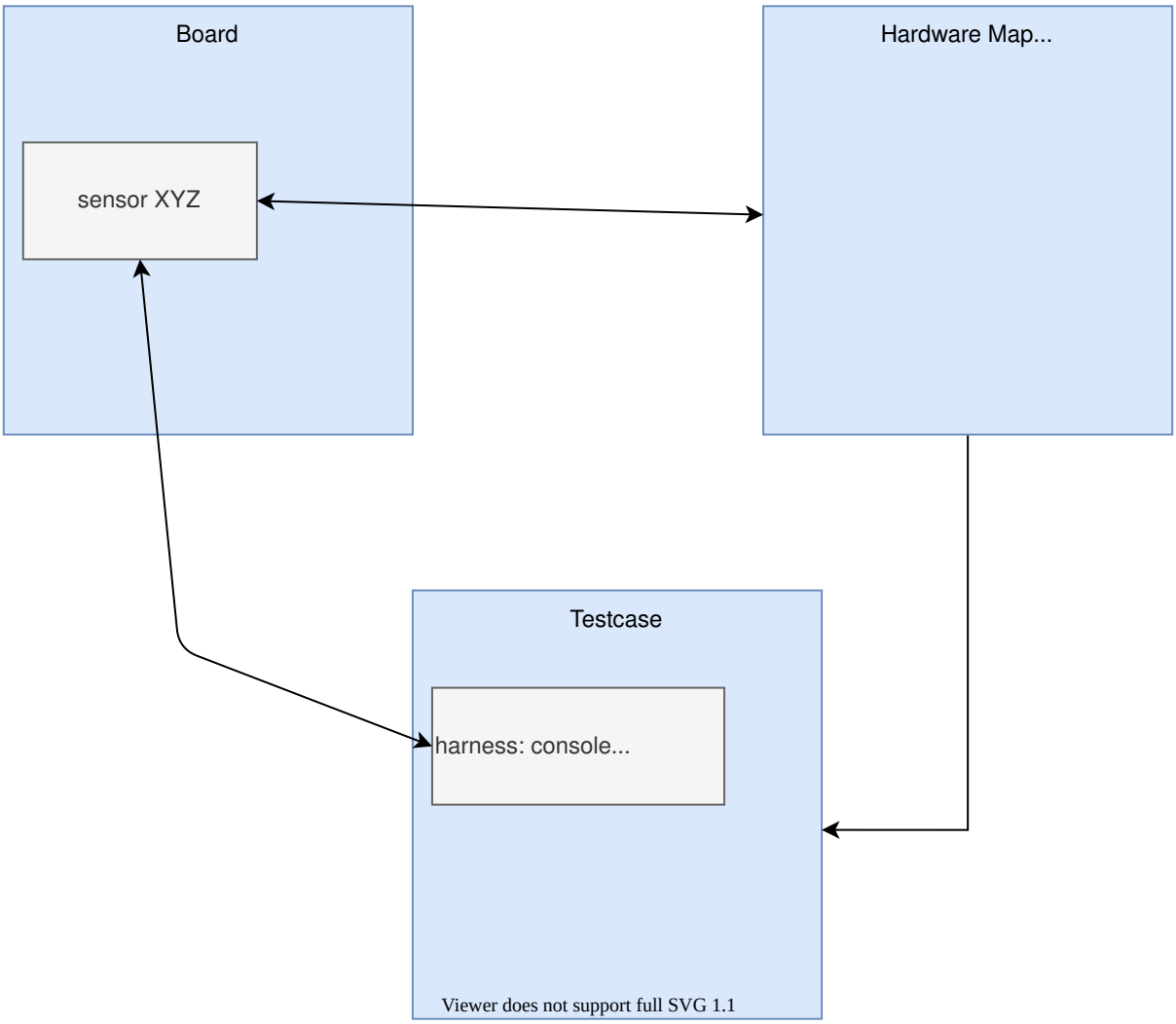
Note: Currently only boards with support for both pyocd and nrfjprog are supported with the hardware map features. Boards that require other runners to flash the Zephyr binary are still work in progress.

Fixtures Some tests require additional setup or special wiring specific to the test. Running the tests without this setup or test fixture may fail. A testcase can specify the fixture it needs which can then be matched with hardware capability of a board and the fixtures it supports via the command line or using the hardware map file.

Fixtures are defined in the hardware map file as a list:

```
- connected: true
  fixtures:
    - gpio_loopback
  id: 0240000026334e450015400f5e0e000b4eb1000097969900
  platform: frdm_k64f
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/ttyACM9
```

When running *Test Runner (Twister)* with `--device-testing`, the configured fixture in the hardware map file will be matched to testcases requesting the same fixtures and these tests will be executed on the boards that provide this fixture.



Notes It may be useful to annotate board descriptions in the hardware map file with additional information. Use the "notes" keyword to do this. For example:

```
- connected: false
  fixtures:
    - gpio_loopback
  id: 000683290670
  notes: An nrf5340dk_nrf5340 is detected as an nrf52840dk_nrf52840 with no serial
    port, and three serial ports with an unknown platform.  The board id of the serial
    ports is not the same as the board id of the the development kit.  If you␣
↪regenerate
    this file you will need to update serial to reference the third port, and platform
    to nrf5340dk_nrf5340_cpuapp or another supported board target.
  platform: nrf52840dk_nrf52840
  product: J-Link
  runner: jlink
  serial: null
```

Overriding Board Identifier When (re-)generated the hardware map file will contain an "id" keyword that serves as the argument to `--board-id` when flashing. In some cases the detected ID is not the correct one to use, for example when using an external J-Link probe. The "probe_id" keyword overrides the "id" keyword for this purpose. For example:

```
- connected: false
  id: 0229000005d9ebc60000000000000000000000000097969905
  platform: mimxrt1060_evk
  probe_id: 000609301751
  product: DAPLink CMSIS-DAP
  runner: jlink
  serial: null
```

Quarantine Twister allows using user-defined yaml files defining the list of tests to be put under quarantine. Such tests will be skipped and marked accordingly in the output reports. This feature is especially useful when running larger test suits, where a failure of one test can affect the execution of other tests (e.g. putting the physical board in a corrupted state).

To use the quarantine feature one has to add the argument `--quarantine-list <PATH_TO_QUARANTINE_YAML>` to a twister call. The current status of tests on the quarantine list can also be verified by adding `--quarantine-verify` to the above argument. This will make twister skip all tests which are not on the given list.

A quarantine yaml has to be a sequence of dictionaries. Each dictionary has to have "scenarios" and "platforms" entries listing combinations of scenarios and platforms to put under quarantine. In addition, an optional entry "comment" can be used, where some more details can be given (e.g. link to a reported issue). These comments will also be added to the output reports.

An example of entries in a quarantine yaml:

```
- scenarios:
    - sample.basic.helloworld
  platforms:
    - all
  comment: "Link to the issue: https://github.com/zephyrproject-rtos/zephyr/pull/33287
↪"

- scenarios:
    - kernel.common
    - kernel.common.misra
    - kernel.common.nano64
```

(continues on next page)

(continued from previous page)

```
platforms:
  - qemu_cortex_m3
  - native_posix
```

8.18.3 Generating coverage reports

With Zephyr, you can generate code coverage reports to analyze which parts of the code are covered by a given test or application.

You can do this in two ways:

- In a real embedded target or QEMU, using Zephyr's gcov integration
- Directly in your host computer, by compiling your application targeting the POSIX architecture

Test coverage reports in embedded devices or QEMU

Overview GCC GCOV is a test coverage program used together with the GCC compiler to analyze and create test coverage reports for your programs, helping you create more efficient, faster running code and discovering untested code paths

In Zephyr, gcov collects coverage profiling data in RAM (and not to a file system) while your application is running. Support for gcov collection and reporting is limited by available RAM size and so is currently enabled only for QEMU emulation of embedded targets.

Details There are 2 parts to enable this feature. The first is to enable the coverage for the device and the second to enable in the test application. As explained earlier the code coverage with gcov is a function of RAM available. Therefore ensure that the device has enough RAM when enabling the coverage for it. For example a small device like frdm_k64f can run a simple test application but the more complex test cases which consume more RAM will crash when coverage is enabled.

To enable the device for coverage, select `CONFIG_HAS_COVERAGE_SUPPORT` in the Kconfig.board file.

To report the coverage for the particular test application set `CONFIG_COVERAGE`.

Steps to generate code coverage reports These steps will produce an HTML coverage report for a single application.

1. Build the code with CONFIG_COVERAGE=y.

   ```
   west build -b mps2_an385 -- -DCONFIG_COVERAGE=y
   ```

2. Capture the emulator output into a log file. You may need to terminate the emulator with `Ctrl-A X` for this to complete after the coverage dump has been printed:

   ```
   ninja -Cbuild run | tee log.log
   ```

   or

   ```
   ninja -Cbuild run | tee log.log
   ```

3. Generate the gcov `.gcda` and `.gcno` files from the log file that was saved:

   ```
   $ python3 scripts/gen_gcov_files.py -i log.log
   ```

4. Find the gcov binary placed in the SDK. You will need to pass the path to the gcov binary for the appropriate architecture when you later invoke `gcovr`:

```
$ find $ZEPHYR_SDK_INSTALL_DIR -iregex ".*gcov"
```

5. Create an output directory for the reports:

```
$ mkdir -p gcov_report
```

6. Run `gcovr` to get the reports:

```
$ gcovr -r $ZEPHYR_BASE . --html -o gcov_report/coverage.html --html-details --
↪gcov-executable <gcov_path_in_SDK>
```

Coverage reports using the POSIX architecture

When compiling for the POSIX architecture, you utilize your host native tooling to build a native executable which contains your application, the Zephyr OS, and some basic HW emulation.

That means you can use the same tools you would while developing any other desktop application.

To build your application with `gcc`'s gcov, simply set `CONFIG_COVERAGE` before compiling it. When you run your application, `gcov` coverage data will be dumped into the respective `gcda` and `gcno` files. You may postprocess these with your preferred tools. For example:

```
west build -b native_posix samples/hello_world -- -DCONFIG_COVERAGE=y
```

```
$ ./build/zephyr/zephyr.exe
# Press Ctrl+C to exit
lcov --capture --directory ./ --output-file lcov.info -q --rc lcov_branch_coverage=1
genhtml lcov.info --output-directory lcov_html -q --ignore-errors source --branch-
↪coverage --highlight --legend
```

Note: You need a recent version of lcov (at least 1.14) with support for intermediate text format. Such packages exist in recent Linux distributions.

Alternatively, you can use gcovr (at least version 4.2).

Coverage reports using Twister

Zephyr's *twister script* can automatically generate a coverage report from the tests which were executed. You just need to invoke it with the `--coverage` command line option.

For example, you may invoke:

```
$ twister --coverage -p qemu_x86 -T tests/kernel
```

or:

```
$ twister --coverage -p native_posix -T tests/bluetooth
```

which will produce `twister-out/coverage/index.html` with the report.

The process differs for unit tests, which are built with the host toolchain and require a different board:

```
$ twister --coverage -p unit_testing -T tests/unit
```

which produces a report in the same location as non-unit testing.

8.19 Trusted Firmware-M

8.19.1 Trusted Firmware-M Overview

Trusted Firmware-M (TF-M) is a reference implementation of the Platform Security Architecture (PSA) IoT Security Framework. It defines and implements an architecture and a set of software components that aim to address some of the main security concerns in IoT products.

Zephyr RTOS has been PSA Certified since Zephyr 2.0.0 with TF-M 1.0, and is currently integrated with TF-M 1.3.0.

What Does TF-M Offer?

Through a set of secure services and by design, TF-M provides:

- Isolation of secure and non-secure resources
- Embedded-appropriate crypto
- Management of device secrets (keys, etc.)
- Firmware verification (and encryption)
- Protected off-chip data storage and retrieval
- Proof of device identity (device attestation)
- Audit logging

Build System Integration

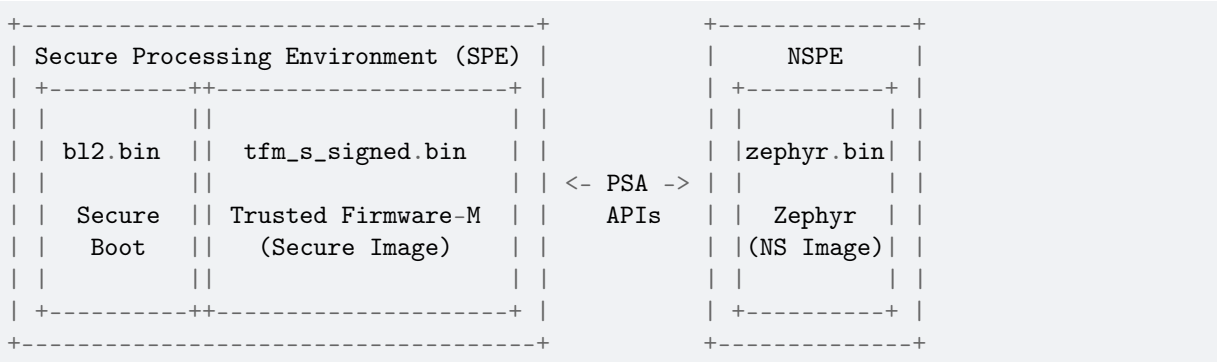
When using TF-M with a supported platform, TF-M will be automatically built and link in the background as part of the standard Zephyr build process. This build process makes a number of assumptions about how TF-M is being used, and has certain implications about what the Zephyr application image can and can not do:

- The secure processing environment (secure boot and TF-M) starts first
- Resource allocation for Zephyr relies on choices made in the secure image.

Architecture Overview

A TF-M application will, generally, have the following three parts, from most to least trusted, left-to-right, with code execution happening in the same order (secure boot > secure image > ns image).

While the secure bootloader is optional, it is enabled by default, and secure boot is an important part of providing a secure solution:



Communication between the (Zephyr) Non-Secure Processing Environment (NSPE) and the (TF-M) Secure Processing Environment image happens based on a set of PSA APIs, and normally makes use of an IPC mechanism that is included as part of the TF-M build, and implemented in Zephyr (see modules/trusted-firmware-m/interface).

Root of Trust (RoT) Architecture TF-M is based upon a **Root of Trust (RoT)** architecture. This allows for hierarchies of trust from most, to less, to least trusted, providing a sound foundation upon which to build or access trusted services and resources.

The benefit of this approach is that less trusted components are prevented from accessing or compromising more critical parts of the system, and error conditions in less trusted environments won't corrupt more trusted, isolated resources.

The following RoT hierarchy is defined for TF-M, from most to least trusted:

- PSA Root of Trust (**PRoT**), which consists of:
  - PSA Immutable Root of Trust: secure boot
  - PSA Updateable Root of Trust: most trusted secure services
- Application Root of Trust (**ARoT**): isolated secure services

The **PSA Immutable Root of Trust** is the most trusted piece of code in the system, to which subsequent Roots of Trust are anchored. In TF-M, this is the secure boot image, which verifies that the secure and non-secure images are valid, have not been tampered with, and come from a reliable source. The secure bootloader also verifies new images during the firmware update process, thanks to the public signing key(s) built into it. As the name implies, this image is **immutable**.

The **PSA Updateable Root of Trust** implements the most trusted secure services and components in TF-M, such as the Secure Partition Manager (SPM), and shared secure services like PSA Crypto, Internal Trusted Storage (ITS), etc. Services in the PSA Updateable Root of Trust have access to other resources in the same Root of Trust.

The **Application Root of Trust** is a reduced-privilege area in the secure processing environment which, depending on the isolation level chosen when building TF-M, has limited access to the PRoT, or even other ARoT services at the highest isolation levels. Some standard services exist in the ARoT, such as Protected Storage (PS), and generally custom secure services that you implement should be placed in the ARoT, unless a compelling reason is present to place them in the PRoT.

These divisions are distinct from the **untrusted code**, which runs in the non-secure environment, and has the least privilege in the system. This is the Zephyr application image in this case.

Isolation Levels At present, there are three distinct **isolation levels** defined in TF-M, with increasingly rigid boundaries between regions. The isolation level used will depend on your security requirements, and the system resources available to you.

- **Isolation Level 1** is the lowest isolation level, and the only major boundary is between the secure and non-secure processing environment, usually by means of Arm TrustZone on Armv8-M processors. There is no distinction here between the PSA Updateable Root of Trust (PRoT) and the Application Root of Trust (ARoT). They execute at the same privilege level. This isolation level will lead to the smallest combined application images.
- **Isolation Level 2** builds upon level one by introducing a distinction between the PSA Updateable Root of Trust and the Application Root of Trust, where ARoT services have limited access to PRoT services, and can only communicate with them through public APIs exposed by the PRoT services. ARoT services, however, are not strictly isolated from one another.
- **Isolation Level 3** is the highest isolation level, and builds upon level 2 by isolating ARoT services from each other, so that each ARoT is essentially silo'ed from other services. This provides the highest level of isolation, but also comes at the cost of additional overhead and code duplication between services.

The current isolation level can be checked via `CONFIG_TFM_ISOLATION_LEVEL`.

Secure Boot The default secure bootloader in TF-M is based on MCUBoot, and is referred to as `BL2` in TF-M (for the second-stage bootloader, potentially after a HW-based bootloader on the secure MCU, etc.).

All images in TF-M are hashed and signed, with the hash and signature verified by MCUBoot during the firmware update process.

Some key features of MCUBooot as used in TF-M are:

- Public signing key(s) are baked into the bootloader
- S and NS images can be signed using different keys
- Firmware images can optionally be encyrpted
- Client software is responsible for writing a new image to the secondary slot
- By default, uses static flash layout of two identically-sized memory regions
- Optional security counter for rollback protection

When dealing with (optionally) encrypted images:

- Only the payload is encrypted (header, TLVs are plain text)
- Hashing and signing are applied over the un-encrypted data
- Uses `AES-CTR-128` or `AES-CTR-256` for encryption
- Encryption key randomized every encryption cycle (via `imgtool`)
- The `AES-CTR` key is included in the image and can be encrypted using:
  - `RSA-OAEP`
  - `AES-KW` (128 or 256 bits depending on the `AES-CTR` key length)
  - `ECIES-P256`
  - `ECIES-X25519`

Key config properties to control secure boot in Zephyr are:

- `CONFIG_TFM_BL2` toggles the bootloader (default = `y`).
- `CONFIG_TFM_KEY_FILE_S` overrides the secure signing key.
- `CONFIG_TFM_KEY_FILE_NS` overrides the non-secure signing key.

Secure Processing Environment Once the secure bootloader has finished executing, a TF-M based secure image will begin execution in the **secure processing environment**. This is where our device will be initially configured, and any secure services will be initialised.

Note that the starting state of our device is controlled by the secure firmware, meaning that when the non-secure Zephyr application starts, peripherals may not be in the HW-default reset state. In case of doubts, be sure to consult the board support packages in TF-M, available in the `platform/ext/target/` folder of the TF-M module (which is in `modules/tee/tfm/trusted-firmware-m/` within a default Zephyr west workspace.)

Secure Services As of TF-M 1.3.0, the following secure services are available:

- Audit Logging (Audit)
- Crypto (Crypto)
- Firmware Update (FWU)
- Initial Attestation (IAS)
- Secure Storage, which has two parts:

- Internal Trusted Storage (ITS)
- Protected Storage (PS)

A template also exists for creating your own custom services.

For full details on these services, and their exposed APIs, please consult the TF-M Documentation.

Key Management and Derivation Key and secret management is a critical part of any secure device. You need to ensure that key material is available to regions that require it, but not to anything else, and that it is stored securely in a way that makes it difficult to tamper with or maliciously access.

The **Internal Trusted Storage** service in TF-M is used by the **PSA Crypto** service (which itself makes use of mbedtls) to store keys, and ensure that private keys are only ever accessible to the secure processing environment. Crypto operations that make use of key material, such as when signing payloads or when decrypting sensitive data, all take place via key handles. At no point should the key material ever be exposed to the NS environment.

One exception is that private keys can be provisioned into the secure processing environment as a one-way operation, such as during a factory provisioning process, but even this should be avoided where possible, and a request should be made to the SPE (via the PSA Crypto service) to generate a new private key itself, and the public key for that can be requested during provisioning and logged in the factory. This ensures the private key material is never exposed, or even known during the provisioning phase.

TF-M also makes extensive use of the **Hardware Unique Key (HUK)**, which every TF-M device must provide. This device-unique key is used by the **Protected Storage** service, for example, to encrypt information stored in external memory. For example, this ensures that the contents of flash memory can't be decrypted if they are removed and placed on a new device, since each device has its own unique HUK used while encrypting the memory contents the first time.

HUKs provide an additional advantage for developers, in that they can be used to derive new keys, and the **derived keys** don't need to be stored since they can be regenerated from the HUK at startup, using an additional salt/seed value (depending on the key derivation algorithm used). This removes the storage issue and a frequent attack vector. The HUK itself it usually highly protected in secure devices, and inaccessible directly by users.

`TFM_CRYPTO_ALG_HUK_DERIVATION` identifies the default key derivation algorithm used if a software implementation is used. The current default algorithm is `HKDF` (RFC 5869) with a SHA-256 hash. Other hardware implementations may be available on some platforms.

Non-Secure Processing Environment Zephyr is used for the NSPE, using a board that is supported by TF-M where the `CONFIG_BUILD_WITH_TFM` flag has been enabled.

Generally, you simply need to select the `*_ns` variant of a valid target (for example `mps2_an521_ns`), which will configure your Zephyr application to run in the NSPE, correctly build and link it with the TF-M secure images, sign the secure and non-secure images, and merge the three binaries into a single `tfm_merged.hex` file. The *west flash* command will flash `tfm_merged.hex` by default in this configuration.

At present, Zephyr can not be configured to be used as the secure processing environment.

8.19.2 TF-M Requirements

The following are some of the boards that can be used with TF-M:

Board	NSPE board name
mps2_an521_board	`mps2_an521_ns` (qemu supported)
bl5340_dvk	`bl5340_dvk_cpuapp_ns`
lpcxpresso55s69	`lpcxpresso55s69_ns`
nrf9160dk_nrf9160	`nrf9160dk_nrf9160_ns`
nrf5340dk_nrf5340	`nrf5340dk_nrf5340_cpuapp_ns`
nucleo_l552ze_q_board	`nucleo_l552ze_q_ns`
stm32l562e_dk_board	`stm32l562e_dk_ns`
v2m_musca_b1_board	`v2m_musca_b1_ns`
v2m_musca_s1_board	`v2m_musca_s1_ns`

You can run `west boards -n _ns$` to search for non-secure variants of different board targets. To make sure TF-M is supported for a board in its output, check that `CONFIG_TRUSTED_EXECUTION_NONSECURE` is set to `y` in that board's default configuration.

Software Requirements

The following Python modules are required when building TF-M binaries:

- cryptography
- pyasn1
- pyyaml
- cbor>=1.0.0
- imgtool>=1.6.0
- jinja2
- click

You can install them via:

```
$ pip3 install --user cryptography pyasn1 pyyaml cbor>=1.0.0 imgtool>=1.6.0␣
↪jinja2 click
```

They are used by TF-M's signing utility to prepare firmware images for validation by the bootloader.

Part of the process of generating binaries for QEMU and merging signed secure and non-secure binaries on certain platforms also requires the use of the `srec_cat` utility.

This can be installed on Linux via:

```
$ sudo apt-get install srecord
```

And on OS X via:

```
$ brew install srecord
```

For Windows-based systems, please make sure you have a copy of the utility available on your system path. See, for example: SRecord for Windows

8.19.3 TF-M Build System

When building a valid `_ns` board target, TF-M will be built in the background, and linked with the Zephyr non-secure application. No knowledge of TF-M's build system is required in most cases, and the following will build a TF-M and Zephyr image pair, and run it in qemu with no additional steps required:

```
$ west build -p auto -t mps2_an521_ns samples/tfm_integration/psa_crypto/ -
↪t run
```

The outputs and certain key steps in this build process are described here, however, since you will need to understand and interact with the outputs, and deal with signing the secure and non-secure images before deploying them.

Images Created by the TF-M Build

The TF-M build system creates the following executable files:

- tfm_s - the secure firmware
- tfm_ns - a nonsecure app which is discarded in favor of the Zephyr app
- bl2 - mcuboot, if enabled

For each of these, it creates .bin, .hex, .elf, and .axf files.

The TF-M build system also creates signed variants of tfm_s and tfm_ns, and a file which combines them:

- tfm_s_signed
- tfm_ns_signed
- tfm_s_ns_signed

For each of these, only .bin files are created.

The Zephyr build system usually signs both tfm_s and the Zephyr ns app itself. See below for details.

The 'tfm' target contains properties for all these paths. For example, the following will resolve to `<path>/tfm_s.hex`:

```
$<TARGET_PROPERTY:tfm,TFM_S_HEX_FILE>
```

See the top level CMakeLists.txt file in the tfm module for an overview of all the properties.

Signing Images

When `CONFIG_TFM_BL2` is set to `y`, TF-M uses a secure bootloader (BL2) and firmware images must be signed with a private key. The firmware image is validated by the bootloader during updates using the corresponding public key, which is stored inside the secure bootloader firmware image.

By default, `tfm/bl2/ext/mcuboot/root-rsa-3072.pem` is used to sign secure images, and `tfm/bl2/ext/mcuboot/root-rsa-3072_1.pem` is used to sign non-secure images. Theses default .pem keys can (and **should**) be overridden using the `CONFIG_TFM_KEY_FILE_S` and `CONFIG_TFM_KEY_FILE_NS` config flags.

To satisfy PSA Certified Level 1 requirements, **You MUST replace the default .pem file with a new key pair!**

To generate a new public/private key pair, run the following commands:

```
$ imgtool keygen -k root-rsa-3072_s.pem -t rsa-3072
$ imgtool keygen -k root-rsa-3072_ns.pem -t rsa-3072
```

You can then place the new .pem files in an alternate location, such as your Zephyr application folder, and reference them in the `prj.conf` file via the `CONFIG_TFM_KEY_FILE_S` and `CONFIG_TFM_KEY_FILE_NS` config flags.

Warning: Be sure to keep your private key file in a safe, reliable location! If you lose this key file, you will be unable to sign any future firmware images, and it will no longer be possible to update your devices in the field!

After the built-in signing script has run, it creates a `tfm_merged.hex` file that contains all three binaries: bl2, tfm_s, and the zephyr app. This hex file can then be flashed to your development board or run in QEMU.

Custom CMake arguments When building a Zephyr application with TF-M it might be necessary to control the CMake arguments passed to the TF-M build.

Zephyr TF-M build offers several Kconfig options for controlling the build, but doesn't cover every CMake argument supported by the TF-M build system.

The `TFM_CMAKE_OPTIONS` property on the `zephyr_property_target` can be used to pass custom CMake arguments to the TF-M build system.

To pass the CMake argument `-DFOO=bar` to the TF-M build system, place the following CMake snippet in your CMakeLists.txt file.

```
set_property(TARGET zephyr_property_target
             APPEND PROPERTY TFM_CMAKE_OPTIONS
             -DFOO=bar
)
```

Note: The `TFM_CMAKE_OPTIONS` is a list so it is possible to append multiple options. Also CMake generator expressions are supported, such as `$<1:-DFOO=bar>`

8.19.4 Trusted Firmware-M Integration

The Trusted Firmware-M (TF-M) section contains information about the integration between TF-M and Zephyr RTOS. Use this information to help understand how to integrate TF-M with Zephyr for Cortex-M platforms and make use of its secure run-time services in Zephyr applications.

Board Definitions

TF-M will be built for the secure processing environment along with Zephyr if the `CONFIG_BUILD_WITH_TFM` flag is set to `y`.

Generally, this value should never be set at the application level, however, and all config flags required for TF-M should be set in a board variant with the `_ns` suffix.

This board variant must define an appropriate flash, SRAM and peripheral configuration that takes into account the initialisation process in the secure processing environment. `CONFIG_TFM_BOARD` must also be set via modules/trusted-firmware-m/Kconfig.tfm to the board name that TF-M expects for this target, so that it knows which target to build for the secure processing environment.

Example: `mps2_an521_ns` The `mps2_an521` target is a dual-core Arm Cortex-M33 evaluation board that, when using the default board variant, would generate a secure Zephyr binary.

The optional `mps2_an521_ns` target, however, sets these additional kconfig flags that indicate that Zephyr should be built as a non-secure image, linked with TF-M as an external project, and optionally the secure bootloader:

- `CONFIG_TRUSTED_EXECUTION_NONSECURE y`

- CONFIG_ARM_TRUSTZONE_M y

Comparing the mps2_an521.dts and mps2_an521_ns.dts files, we can see that the _ns version defines offsets in flash and SRAM memory, which leave the required space for TF-M and the secure bootloader:

```
reserved-memory {
            #address-cells = <1>;
            #size-cells = <1>;
            ranges;

            /* The memory regions defined below must match what the TF-M
             * project has defined for that board - a single image boot is
             * assumed. Please see the memory layout in:
             * https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/
↪platform/ext/target/mps2/an521/partition/flash_layout.h
             */

            code: memory@100000 {
                    reg = <0x00100000 DT_SIZE_K(512)>;
            };

            ram: memory@28100000 {
                    reg = <0x28100000 DT_SIZE_M(1)>;
            };
    };
```

This reserves 1 MB of code memory and 1 MB of RAM for secure boot and TF-M, such that our non-secure Zephyr application code will start at 0x10000, with RAM at 0x28100000. 512 KB code memory is available for the NS zephyr image, along with 1 MB of RAM.

This matches the flash memory layout we see in flash_layout.h in TF-M:

```
* 0x0000_0000 BL2 - MCUBoot (0.5 MB)
* 0x0008_0000 Secure image     primary slot (0.5 MB)
* 0x0010_0000 Non-secure image primary slot (0.5 MB)
* 0x0018_0000 Secure image     secondary slot (0.5 MB)
* 0x0020_0000 Non-secure image secondary slot (0.5 MB)
* 0x0028_0000 Scratch area (0.5 MB)
* 0x0030_0000 Protected Storage Area (20 KB)
* 0x0030_5000 Internal Trusted Storage Area (16 KB)
* 0x0030_9000 NV counters area (4 KB)
* 0x0030_A000 Unused (984 KB)
```

mps2/an521 will be passed in to Tf-M as the board target, specified via CONFIG_TFM_BOARD.

8.20 West (Zephyr's meta-tool)

The Zephyr project includes a swiss-army knife command line tool named west[1]. West is developed in its own repository.

West's built-in commands provide a multiple repository management system with features inspired by Google's Repo tool and Git submodules. West is also "pluggable": you can write your own west extension commands which add additional features to west. Zephyr uses this to provide conveniences for building applications, flashing and debugging them, and more.

Like git and docker, the top-level west command takes some common options, a sub-command to run, and then options and arguments for that sub-command:

[1] Zephyr is an English name for the Latin Zephyrus, the ancient Greek god of the west wind.

```
west [common-opts] <command> [opts] <args>
```

Since west v0.8, you can also run west like this:

```
python3 -m west [common-opts] <command> [opts] <args>
```

You can run `west --help` (or `west -h` for short) to get top-level help for available west commands, and `west <command> -h` for detailed help on each command.

The following pages document west's `v0.11.x` releases, and provide additional context about the tool.

8.20.1 Installing west

West is written in Python 3 and distributed through PyPI. Use `pip3` to install or upgrade west:

On Linux:

```
pip3 install --user -U west
```

On Windows and macOS:

```
pip3 install -U west
```

Note: See *Python and pip* for additional clarification on using the `--user` switch.

Afterwards, you can run `pip3 show -f west` for information on where the west binary and related files were installed.

Once west is installed, you can use it to *clone the Zephyr repositories*.

Structure

West's code is distributed via PyPI in a Python package named `west`. This distribution includes a launcher executable, which is also named `west` (or `west.exe` on Windows).

When west is installed, the launcher is placed by `pip3` somewhere in the user's filesystem (exactly where depends on the operating system, but should be on the `PATH` *environment variable*). This launcher is the command-line entry point to running both built-in commmands like `west init`, `west update`, along with any extensions discovered in the workspace.

In addition to its command-line interface, you can also use west's Python APIs directly. See west-apis for details.

Enabling shell completion

West currently supports shell completion in the following combinations of platform and shell:

- Linux: bash
- macOS: bash
- Windows: not available

In order to enable shell completion, you will need to obtain the corresponding completion script and have it sourced every time you enter a new shell session.

To obtain the completion script you can use the `west completion` command:

```
cd /path/to/zephyr/
west completion bash > ~/west-completion.bash
```

Note: Remember to update your local copy of the completion script using `west completion` when you update Zephyr.

Next, you need to import `west-completion.bash` into your bash shell.

On Linux, you have the following options:

- Copy `west-completion.bash` to `/etc/bash_completion.d/`.
- Copy `west-completion.bash` to `/usr/share/bash-completion/completions/`.
- Copy `west-completion.bash` to a local folder and source it from your `~/.bashrc`.

On macOS, you have the following options:

- Copy `west-completion.bash` to a local folder and source it from your `~/.bash_profile`
- Install the `bash-completion` package with `brew`:

  ```
  brew install bash-completion
  ```

  then source the main bash completion script in your `~/.bash_profile`:

  ```
  source /usr/local/etc/profile.d/bash_completion.sh
  ```

  and finally copy `west-completion.bash` to `/usr/local/etc/bash_completion.d/`.

8.20.2 West Release Notes

v0.11.1

New features:

- `west status` now only prints output for projects which have a nonempty status.

Bug fixes:

- The manifest file parser was incorrectly allowing project names which contain the path separator characters `/` and `\`. These invalid characters are now rejected.

  Note: if you need to place a project within a subdirectory of the workspace topdir, use the `path:` key. If you need to customize a project's fetch URL relative to its remote `url-base:`, use `repo-path:`. See *Projects* for examples.

- The changes made in west v0.10.1 to the `west init --manifest-rev` option which selected the default branch name were leaving the manifest repository in a detached HEAD state. This has been fixed by using `git clone` internally instead of `git init` and `git fetch`. See issue #522 for details.

- The `WEST_CONFIG_LOCAL` environment variable now correctly overrides the default location, `<workspace topdir>/.west/config`.

- `west update --fetch=smart` (`smart` is the default) now correctly skips fetches for project revisions which are lightweight tags (it already worked correctly for annotated tags; only lightweight tags were unnecessarily fetched).

Other changes:

- The fix for issue #522 mentioned above introduces a new restriction. The `west init --manifest-rev` option value, if given, must now be either a branch or a tag. In particular, "pseudo-branches" like GitHub's `pull/1234/head` references which could previously be used to fetch a pull request can no longer be passed to `--manifest-rev`. Users must now fetch and check out such revisions manually after running `west init`.

API changes:

- `west.manifest.Manifest.get_projects()` avoids incorrect results in some edge cases described in issue #523.
- `west.manifest.Project.sha()` now works correctly for tag revisions. (This applies to both lightweight and annotated tags.)

v0.11.0

New features:

- `west update` now supports `--narrow`, `--name-cache`, and `--path-cache` options. These can be influenced by the `update.narrow`, `update.name-cache`, and `update.path-cache` *Configuration* options. These can be used to optimize the speed of the update.
- `west update` now supports a `--fetch-opt` option that will be passed to the `git fetch` command used to fetch remote revisions when updating each project.

Bug fixes:

- `west update` now synchronizes Git submodules in projects by default. This avoids issues if the URL changes in the manifest file from when the submodule was first initialized. This behavior can be disabled by setting the `update.sync-submodules` configuration option to `false`.

Other changes:

- the west-apis-manifest module has fixed docstrings for the Project class

v0.10.1

New features:

- The *west init* command's `--manifest-rev` (`--mr`) option no longer defaults to `master`. Instead, the command will query the repository for its default branch name and use that instead. This allows users to move from `master` to `main` without breaking scripts that do not provide this option.

v0.10.0

New features:

- The `name` key in a project's *submodules list* is now optional.

Bug fixes:

- West now checks that the manifest schema version is one of the explicitly allowed vlaues documented in *Version*. The old behavior was just to check that the schema version was newer than the west version where the `manifest: version:` key was introduced. This incorrectly allowed invalid schema versions, like `0.8.2`.

Other changes:

- A manifest file's `group-filter` is now propagated through an `import`. This is a change from how west v0.9.x handled this. In west v0.9.x, only the top level manifest file's `group-filter` had any effect; the group filter lists from any imported manifests were ignored.

Starting with west v0.10.0, the group filter lists from imported manifests are also imported. For details, see *Group Filters and Imports*.

The new behavior will take effect if `manifest: version:` is not given or is at least `0.10`. The old behavior is still available in the top level manifest file only with an explicit `manifest: version: 0.9`. See *Version* for more information on schema versions.

See west pull request #482 for the motivation for this change and additional context.

v0.9.1

Bug fixes:

- Commands like `west manifest --resolve` now correctly include group and group filter information.

Other changes:

- West now warns if you combine `import` with `group-filter`. Semantics for this combination have changed starting with v0.10.x. See the v0.10.0 release notes above for more information.

v0.9.0

Warning: The `west config` fix described below comes at a cost: any comments or other manual edits in configuration files will be removed when setting a configuration option via that command or the `west.configuration` API.

Warning: Combining the `group-filter` feature introduced in this release with manifest imports is discouraged. The resulting behavior has changed in west v0.10.

New features:

- West manifests now support *Git Submodules in Projects*. This allows you to clone Git submodules into a west project repository in addition to the project repository itself.
- West manifests now support *Project Groups and Active Projects*. Project groups can be enabled and disabled to determine what projects are "active", and therefore will be acted upon by the following commands: `west update`, `west list`, `west diff`, `west status`, `west forall`.
- `west update` no longer updates inactive projects by default. It now supports a `--group-filter` option which allows for one-time modifications to the set of enabled and disabled project groups.
- Running `west list`, `west diff`, `west status`, or `west forall` with no arguments does not print information for inactive projects by default. If the user specifies a list of projects explicitly at the command line, output for them is included regardless of whether they are active.

  These commands also now support `--all` arguments to include all projects, even inactive ones.
- `west list` now supports a `{groups}` format string key in its `--format` argument.

Bug fixes:

- The `west config` command and `west.configuration` API did not correctly store some configuration values, such as strings which contain commas. This has been fixed; see commit 36f3f91e for details.
- A manifest file with an empty `manifest: self: path:` value is invalid, but west used to let it pass silently. West now rejects such manifests.
- A bug affecting the behavior of the `west init -l .` command was fixed; see issue #435.

API changes:

- added `west.manifest.Manifest.is_active()`
- added `west.manifest.Manifest.group_filter`
- added `submodules` attribute to `west.manifest.Project`, which has newly added type `west.manifest.Submodule`

Other changes:

- The *Manifest Imports* feature now supports the terms `allowlist` and `blocklist` instead of `whitelist` and `blacklist`, respectively.

  The old terms are still supported for compatibility, but the documentation has been updated to use the new ones exclusively.

v0.8.0

This is a feature release which changes the manifest schema by adding support for a `path-prefix:` key in an `import:` mapping, along with some other features and fixes.

- Manifest import mappings now support a `path-prefix:` key, which places the project and its imported repositories in a subdirectory of the workspace. See *Example 3.4: Import into a subdirectory* for an example.
- The west command line application can now also be run using `python3 -m west`. This makes it easier to run west under a particular Python interpreter without modifying the `PATH` environment variable.
- *west manifest –path* prints the absolute path to west.yml
- `west init` now supports an `--mf foo.yml` option, which initializes the workspace using `foo.yml` instead of `west.yml`.
- `west list` now prints the manifest repository's path using the `manifest.path` *configuration option*, which may differ from the `self: path:` value in the manifest data. The old behavior is still available, but requires passing a new `--manifest-path-from-yaml` option.
- Various Python API changes; see west-apis for details.

v0.7.3

This is a bugfix release.

- Fix an error where a failed import could leave the workspace in an unusable state (see [PR #415](https://github.com/zephyrproject-rtos/west/pull/415) for details)

v0.7.2

This is a bugfix and minor feature release.

- Filter out duplicate extension commands brought in by manifest imports
- Fix `west.Manifest.get_projects()` when finding the manifest repository by path

v0.7.1

This is a bugfix and minor feature release.

- `west update --stats` now prints timing for operations which invoke a subprocess, time spent in west's Python process for each project, and total time updating each project.

- `west topdir` always prints a POSIX style path
- minor console output changes

v0.7.0

The main user-visible feature in west 0.7 is the *Manifest Imports* feature. This allows users to load west manifest data from multiple different files, resolving the results into a single logical manifest.

Additional user-visible changes:

- The idea of a "west installation" has been renamed to "west workspace" in this documentation and in the west API documentation. The new term seems to be easier for most people to work with than the old one.
- West manifests now support a *schema version*.
- The "west config" command can now be run outside of a workspace, e.g. to run `west config --global section.key value` to set a configuration option's value globally.
- There is a new *west topdir* command, which prints the root directory of the current west workspace.
- The `west -vv init` command now prints the git operations being performed, and their results.
- The restriction that no project can be named "manifest" is now enforced; the name "manifest" is reserved for the manifest repository, and is usable as such in commands like `west list manifest`, instead of `west list path-to-manifest-repository` being the only way to say that
- It's no longer an error if there is no project named "zephyr". This is part of an effort to make west generally usable for non-Zephyr use cases.
- Various bug fixes.

The developer-visible changes to the west-apis are:

- west.build and west.cmake: deprecated; this is Zephyr-specific functionality and should never have been part of west. Since Zephyr v1.14 LTS relies on it, it will continue to be included in the distribution, but will be removed when that version of Zephyr is obsoleted.
- west.commands:
  - WestCommand.requires_installation: deprecated; use requires_workspace instead
  - WestCommand.requires_workspace: new
  - WestCommand.has_manifest: new
  - WestCommand.manifest: this is now settable
- west.configuration: callers can now identify the workspace directory when reading and writing configuration files
- west.log:
  - msg(): new
- west.manifest:
  - The module now uses the standard logging module instead of west.log
  - QUAL_REFS_WEST: new
  - SCHEMA_VERSION: new
  - Defaults: removed
  - Manifest.as_dict(): new
  - Manifest.as_frozen_yaml(): new
  - Manifest.as_yaml(): new

  - Manifest.from_file() and from_data(): these factory methods are more flexible to use and less reliant on global state
  - Manifest.validate(): new
  - ManifestImportFailed: new
  - ManifestProject: semi-deprecated and will likely be removed later.
  - Project: the constructor now takes a topdir argument
  - Project.format() and its callers are removed. Use f-strings instead.
  - Project.name_and_path: new
  - Project.remote_name: new
  - Project.sha() now captures stderr
  - Remote: removed

West now requires Python 3.6 or later. Additionally, some features may rely on Python dictionaries being insertion-ordered; this is only an implementation detail in CPython 3.6, but is is part of the language specification as of Python 3.7.

v0.6.3

This point release fixes an error in the behavior of the deprecated `west.cmake` module.

v0.6.2

This point release fixes an error in the behavior of `west update --fetch=smart`, introduced in v0.6.1.

All v0.6.1 users must upgrade.

v0.6.1

Warning: Do not use this point release. Make sure to use v0.6.2 instead.

The user-visible features in this point release are:

- The *west update* command has a new `--fetch` command line flag and `update.fetch` *configuration option*. The default value, "smart", skips fetching SHAs and tags which are available locally.
- Better and more consistent error-handling in the `west diff`, `west status`, `west forall`, and `west update` commands. Each of these commands can operate on multiple projects; if a subprocess related to one project fails, these commands now continue to operate on the rest of the projects. All of them also now report a nonzero error code from the west process if any of these subprocesses fails (this was previously not true of `west forall` in particular).
- The *west manifest* command also handles errors better.
- The *west list* command now works even when the projects are not cloned, as long as its format string only requires information which can be read from the manifest file. It still fails if the format string requires data stored in the project repository, e.g. if it includes the `{sha}` format string key.
- Commands and options which operate on git revisions now accept abbreviated SHAs. For example, `west init --mr SHA_PREFIX` now works. Previously, the `--mr` argument needed to be the entire 40 character SHA if it wasn't a branch or a tag.

The developer-visible changes to the west-apis are:

- west.log.banner(): new
- west.log.small_banner(): new
- west.manifest.Manifest.get_projects(): new
- west.manifest.Project.is_cloned(): new
- west.commands.WestCommand instances can now access the parsed Manifest object via a new self.manifest property during the do_run() call. If read, it returns the Manifest object or aborts the command if it could not be parsed.
- west.manifest.Project.git() now has a capture_stderr kwarg

v0.6.0

- No separate bootstrapper

  In west v0.5.x, the program was split into two components, a bootstrapper and a per-installation clone. See Multiple Repository Management in the v1.14 documentation for more details.

  This is similar to how Google's Repo tool works, and lets west iterate quickly at first. It caused confusion, however, and west is now stable enough to be distributed entirely as one piece via PyPI.

  From v0.6.x onwards, all of the core west commands and helper classes are part of the west package distributed via PyPI. This eliminates complexity and makes it possible to import west modules from anywhere in the system, not just extension commands.

- The `selfupdate` command still exists for backwards compatibility, but now simply exits after printing an error message.
- Manifest syntax changes
  - A west manifest file's `projects` elements can now specify their fetch URLs directly, like so:

    ```
    manifest:
      projects:
        - name: example-project-name
          url: https://github.com/example/example-project
    ```

    Project elements with `url` attributes set in this way may not also have `remote` attributes.
  - Project names must be unique: this restriction is needed to support future work, but was not possible in west v0.5.x because distinct projects may have URLs with the same final pathname component, like so:

    ```
    manifest:
      remotes:
        - name: remote-1
          url-base: https://github.com/remote-1
        - name: remote-2
          url-base: https://github.com/remote-2
      projects:
        - name: project
          remote: remote-1
          path: remote-1-project
        - name: project
          remote: remote-2
          path: remote-2-project
    ```

    These manifests can now be written with projects that use `url` instead of `remote`, like so:

```
manifest:
  projects:
    - name: remote-1-project
      url: https://github.com/remote-1/project
    - name: remote-2-project
      url: https://github.com/remote-2/project
```

- The `west list` command now supports a `{sha}` format string key
- The default format string for `west list` was changed to `"{name:12} {path:28} {revision:40} {url}"`.
- The command `west manifest --validate` can now be run to load and validate the current manifest file, among other error-handling fixes related to manifest parsing.
- Incompatible API changes were made to west's APIs. Further changes are expected until API stability is declared in west v1.0.
  - The `west.manifest.Project` constructor's `remote` and `defaults` positional arguments are now kwargs. A new `url` kwarg was also added; if given, the `Project` URL is set to that value, and the `remote` kwarg is ignored.
  - `west.manifest.MANIFEST_SECTIONS` was removed. There is only one section now, namely `manifest`. The *sections* kwargs in the `west.manifest.Manifest` factory methods and constructor were also removed.
  - The `west.manifest.SpecialProject` class was removed. Use `west.manifest.ManifestProject` instead.

v0.5.x

West v0.5.x is the first version used widely by the Zephyr Project as part of its v1.14 Long-Term Support (LTS) release. The west v0.5.x documentation is available as part of the Zephyr's v1.14 documentation.

West's main features in v0.5.x are:

- Multiple repository management using Git repositories, including self-update of west itself
- Hierarchical configuration files
- Extension commands

Versions Before v0.5.x

Tags in the west repository before v0.5.x are prototypes which are of historical interest only.

8.20.3 Troubleshooting West

This page covers common issues with west and how to solve them.

`west update` fetching failures

One good way to troubleshoot fetching issues is to run `west update` in verbose mode, like this:

```
west -v update
```

The output includes Git commands run by west and their outputs. Look for something like this:

```
=== updating your_project (path/to/your/project):
west.manifest: your_project: checking if cloned
[...other west.manifest logs...]
--- your_project: fetching, need revision SOME_SHA
west.manifest: running 'git fetch ... https://github.com/your-username/your_project ..
↪.' in /some/directory
```

The `git fetch` command example in the last line above is what needs to succeed.

One strategy is to go to `/some/directory`, copy/paste and run the entire `git fetch` command, then debug from there using the documentation for your credential storage helper.

If you're behind a corporate firewall and may have proxy or other issues, `curl -v FETCH_URL` (for HTTPS URLs) or `ssh -v FETCH_URL` (for SSH URLs) may be helpful.

If you can get the `git fetch` command to run successfully without prompting for a password when you run it directly, you will be able to run `west update` without entering your password in that same shell.

"'west' is not recognized as an internal or external command, operable program or batch file.'

On Windows, this means that either west is not installed, or your `PATH` environment variable does not contain the directory where pip installed `west.exe`.

First, make sure you've installed west; see *Installing west*. Then try running `west` from a new `cmd.exe` window. If that still doesn't work, keep reading.

You need to find the directory containing `west.exe`, then add it to your `PATH`. (This `PATH` change should have been done for you when you installed Python and pip, so ordinarily you should not need to follow these steps.)

Run this command in `cmd.exe`:

```
pip3 show west
```

Then:

1. Look for a line in the output that looks like `Location: C:\foo\python\python38\lib\site-packages`. The exact location will be different on your computer.
2. Look for a file named `west.exe` in the `scripts` directory `C:\foo\python\python38\scripts`.

   Important: Notice how `lib\site-packages` in the `pip3 show` output was changed to `scripts`!

   ---

3. If you see `west.exe` in the `scripts` directory, add the full path to `scripts` to your `PATH` using a command like this:

   ```
   setx PATH "%PATH%;C:\foo\python\python38\scripts"
   ```

   Do not just copy/paste this command. The `scripts` directory location will be different on your system.
4. Close your `cmd.exe` window and open a new one. You should be able to run `west`.

"Error: unexpected keyword argument 'requires_workspace'"

This error occurs on some Linux distributions after upgrading to west 0.7.0 or later from 0.6.x. For example:

```
$ west update
[... stack trace ...]
TypeError: __init__() got an unexpected keyword argument 'requires_workspace'
```

This appears to be a problem with the distribution's pip; see this comment in west issue 373 for details. Some versions of **Ubuntu** and **Linux Mint** are known to have this problem. Some users report issues on Fedora as well.

Neither macOS nor Windows users have reported this issue. There have been no reports of this issue on other Linux distributions, like Arch Linux, either.

Workaround 1: remove the old version, then upgrade:

```
$ pip3 show west | grep Location: | cut -f 2 -d ' '
/home/foo/.local/lib/python3.6/site-packages
$ rm -r /home/foo/.local/lib/python3.6/site-packages/west
$ pip3 install --user west==0.7.0
```

Workaround 2: install west in a Python virtual environment

One option is to use the venv module that's part of the Python 3 standard library. Some distributions remove this module from their base Python 3 packages, so you may need to do some additional work to get it installed on your system.

"invalid choice: 'build'" (or 'flash', etc.)

If you see an unexpected error like this when trying to run a Zephyr extension command (like *west flash*, *west build*, etc.):

```
$ west build [...]
west: error: argument <command>: invalid choice: 'build' (choose from 'init', [...])

$ west flash [...]
west: error: argument <command>: invalid choice: 'flash' (choose from 'init', [...])
```

The most likely cause is that you're running the command outside of a *west workspace*. West needs to know where your workspace is to find *Extensions*.

To fix this, you have two choices:

1. Run the command from inside a workspace (e.g. the `zephyrproject` directory you created when you *got started*).

   For example, create your build directory inside the workspace, or run `west flash --build-dir YOUR_BUILD_DIR` from inside the workspace.

2. Set the `ZEPHYR_BASE` *environment variable* and re-run the west extension command. If set, west will use `ZEPHYR_BASE` to find your workspace.

If you're unsure whether a command is built-in or an extension, run `west help` from inside your workspace. The output prints extension commands separately, and looks like this for mainline Zephyr:

```
$ west help

built-in commands for managing git repositories:
  init:                 create a west workspace
  [...]

other built-in commands:
  help:                 get help for west or a command
  [...]
```

(continues on next page)

(continued from previous page)

```
extension commands from project manifest (path: zephyr):
  build:                compile a Zephyr application
  flash:                flash and run a binary on a board
  [...]
```

"invalid choice: 'post-init'"

If you see this error when running `west init`:

```
west: error: argument <command>: invalid choice: 'post-init'
(choose from 'init', 'update', 'list', 'manifest', 'diff',
'status', 'forall', 'config', 'selfupdate', 'help')
```

Then you have an old version of west installed, and are trying to use it in a workspace that requires a more recent version.

The easiest way to resolve this issue is to upgrade west and retry as follows:

1. Install the latest west with the `-U` option for `pip3 install` as shown in *Installing west*.
2. Back up any contents of `zephyrproject/.west/config` that you want to save. (If you don't have any configuration options set, it's safe to skip this step.)
3. Completely remove the `zephyrproject/.west` directory (if you don't, you will get the "already in a workspace" error message discussed next).
4. Run `west init` again.

"already in an installation"

You may see this error when running `west init` with west 0.6:

```
FATAL ERROR: already in an installation (<some directory>), aborting
```

If this is unexpected and you're really trying to create a new west workspace, then it's likely that west is using the `ZEPHYR_BASE` *environment variable* to locate a workspace elsewhere on your system.

This is intentional; it allows you to put your Zephyr applications in any directory and still use west to build, flash, and debug them, for example.

To resolve this issue, unset `ZEPHYR_BASE` and try again.

8.20.4 Basics

This page introduces west's basic concepts and provides references to further reading.

West's built-in commands allow you to work with *projects* (Git repositories) under a common *workspace* directory.

Example workspace

If you've followed the upstream Zephyr getting started guide, your workspace looks like this:

```
zephyrproject/                  # west topdir
├── .west/                      # marks the location of the topdir
│   └── config                  # per-workspace local configuration file
│
│   # The manifest repository, never modified by west after creation:
├── zephyr/                     # .git/ repo
│   ├── west.yml                # manifest file
│   └── [... other files ...]
│
│   # Projects managed by west:
├── modules/
│   └── lib/
│       └── tinycbor/           # .git/ project
├── net-tools/                  # .git/ project
└── [ ... other projects ...]
```

Workspace concepts

Here are the basic concepts you should understand about this structure. Additional details are in *Workspaces*.

topdir Above, `zephyrproject` is the name of the workspace's top level directory, or *topdir*. (The name `zephyrproject` is just an example – it could be anything, like `z`, `my-zephyr-workspace`, etc.)

You'll typically create the topdir and a few other files and directories using *west init*.

.west directory The topdir contains the `.west` directory. When west needs to find the topdir, it searches for `.west`, and uses its parent directory. The search starts from the current working directory (and starts again from the location in the `ZEPHYR_BASE` environment variable as a fallback if that fails).

configuration file The file `.west/config` is the workspace's *local configuration file*.

manifest repository Every west workspace contains exactly one *manifest repository*, which is a Git repository containing a *manifest file*. The location of the manifest repository is given by the *manifest.path configuration option* in the local configuration file.

For upstream Zephyr, `zephyr` is the manifest repository, but you can configure west to use any Git repository in the workspace as the manifest repository. The only requirement is that it contains a valid manifest file. See *Topologies supported* for information on other options, and *West Manifests* for details on the manifest file format.

manifest file The manifest file is a YAML file that defines *projects*, which are the additional Git repositories in the workspace managed by west. The manifest file is named `west.yml` by default; this can be overridden using the `manifest.file` local configuration option.

You use the *west update* command to update the workspace's projects based on the contents of the manifest file.

projects Projects are Git repositories managed by west. Projects are defined in the manifest file and can be located anywhere inside the workspace. In the above example workspace, `tinycbor` and `net-tools` are projects.

By default, the Zephyr *build system* uses west to get the locations of all the projects in the workspace, so any code they contain can be used as *Modules (External projects)*.

extensions Any repository known to west (either the manifest repository or any project repository) can define *Extensions*. Extensions are extra west commands you can run when using that workspace.

The zephyr repository uses this feature to provide Zephyr-specific commands like *west build*. Defining these as extensions keeps west's core agnostic to the specifics of any workspace's Zephyr version, etc.

ignored files A workspace can contain additional Git repositories or other files and directories not managed by west. West basically ignores anything in the workspace except `.west`, the manifest repository, and the projects specified in the manifest file.

west init and west update

The two most important workspace-related commands are `west init` and `west update`.

`west init` **basics** This command creates a west workspace.

Important: West doesn't change your manifest repository contents after `west init` is run. Use ordinary Git commands to pull new versions, etc.

You will typically run it once, like this:

```
west init -m https://github.com/zephyrproject-rtos/zephyr --mr v2.5.0 zephyrproject
```

This will:

1. Create the topdir, `zephyrproject`, along with `.west` and `.west/config` inside it
2. Clone the manifest repository from https://github.com/zephyrproject-rtos/zephyr, placing it into `zephyrproject/zephyr`
3. Check out the `v2.5.0` git tag in your local zephyr clone
4. Set `manifest.path` to `zephyr` in `.west/config`
5. Set `manifest.file` to `west.yml`

Your workspace is now almost ready to use; you just need to run `west update` to clone the rest of the projects into the workspace to finish.

For more details, see *west init*.

`west update` **basics** This command makes sure your workspace contains Git repositories matching the projects in the manifest file.

Important: Whenever you check out a different revision in your manifest repository, you should run `west update` to make sure your workspace contains the project repositories the new revision expects.

The `west update` command reads the manifest file's contents by:

1. Finding the topdir. In the `west init` example above, that means finding `zephyrproject`.
2. Loading `.west/config` in the topdir to read the `manifest.path` (e.g. `zephyr`) and `manifest.file` (e.g. `west.yml`) options.
3. Loading the manifest file given by these options (e.g. `zephyrproject/zephyr/west.yml`).

It then uses the manifest file to decide where missing projects should be placed within the workspace, what URLs to clone them from, and what Git revisions should be checked out locally. Project repositories which already exist are updated in place by fetching and checking out their respective Git revisions in the manifest file.

For more details, see *west update*.

Other built-in commands

See *Built-in commands*.

Zephyr Extensions

See the following pages for information on Zephyr's extension commands:

- *Building, Flashing and Debugging*
- *Signing Binaries*
- *Additional Zephyr extension commands*
- *Enabling shell completion*

Troubleshooting

See *Troubleshooting West*.

8.20.5 Built-in commands

This page describes west's built-in commands, some of which were introduced in *Basics*, in more detail.

Some commands are related to Git commands with the same name, but operate on the entire workspace. For example, `west diff` shows local changes in multiple Git repositories in the workspace.

Some commands take projects as arguments. These arguments can be project names as specified in the manifest file, or (as a fallback) paths to them on the local file system. Omitting project arguments to commands which accept them (such as `west list`, `west forall`, etc.) usually defaults to using all projects in the manifest file plus the manifest repository itself.

For additional help, run `west <command> -h` (e.g. `west init -h`).

west init

This command creates a west workspace. It can be used in two ways:

1. Cloning a new manifest repository from a remote URL
2. Creating a workspace around an existing local manifest repository

Option 1: to clone a new manifest repository from a remote URL, use:

```
west init [-m URL] [--mr REVISION] [--mf FILE] [directory]
```

The new workspace is created in the given `directory`, creating a new `.west` inside this directory. You can give the manifest URL using the `-m` switch, the initial revision to check out using `--mr`, and the location of the manifest file within the repository using `--mf`.

For example, running:

```
west init -m https://github.com/zephyrproject-rtos/zephyr --mr v1.14.0 zp
```

would clone the upstream official zephyr repository into `zp/zephyr`, and check out the `v1.14.0` release. This command creates `zp/.west`, and set the `manifest.path` *configuration option* to `zephyr` to record the location of the manifest repository in the workspace. The default manifest file location is used.

The `-m` option defaults to `https://github.com/zephyrproject-rtos/zephyr`. The `--mf` option defaults to `west.yml`. Since west v0.10.1, west will use the default branch in the manifest repository unless the `--mr` option is used to override it. (In prior versions, `--mr` defaulted to `master`.)

If no `directory` is given, the current working directory is used.

Option 2: to create a workspace around an existing local manifest repository, use:

```
west init -l [--mf FILE] directory
```

This creates `.west` **next to** `directory` in the file system, and sets `manifest.path` to `directory`.

As above, `--mf` defaults to `west.yml`.

Reconfiguring the workspace:

If you change your mind later, you are free to change `manifest.path` and `manifest.file` using *west config* after running `west init`. Just be sure to run `west update` afterwards to update your workspace to match the new manifest file.

west update

```
west update [-f {always,smart}] [-k] [-r]
            [--group-filter FILTER] [--stats] [PROJECT ...]
```

Which projects are updated:

By default, this command parses the manifest file, usually `west.yml`, and updates each project specified there. If your manifest uses *project groups*, then only the active projects are updated.

To operate on a subset of projects only, give `PROJECT` argument(s). Each `PROJECT` is either a project name as given in the manifest file, or a path that points to the project within the workspace. If you specify projects explicitly, they are updated regardless of whether they are active.

Project update procedure:

For each project that is updated, this command:

1. Initializes a local Git repository for the project in the workspace, if it does not already exist
2. Inspects the project's `revision` field in the manifest, and fetches it from the remote if it is not already available locally
3. Sets the project's *manifest-rev* branch to the commit specified by the revision in the previous step
4. Checks out `manifest-rev` in the local working copy as a detached HEAD
5. If the manifest file specifies a *submodules* key for the project, recursively updates the project's submodules as described below.

To avoid unnecessary fetches, `west update` will not fetch project `revision` values which are Git SHAs or tags that are already available locally. This is the behavior when the `-f` (`--fetch`) option has its default value, `smart`. To force this command to fetch from project remotes even if the revisions appear to be available locally, either use `-f always` or set the `update.fetch` *configuration option* to `always`. SHAs may be given as unique prefixes as long as they are acceptable to Git[1].

If the project `revision` is a Git ref that is neither a tag nor a SHA (i.e. if the project is tracking a branch), `west update` always fetches, regardless of `-f` and `update.fetch`.

Some branch names might look like short SHAs, like `deadbeef`. West treats these like SHAs. You can disambiguate by prefixing the `revision` value with `refs/heads/`, e.g. `revision: refs/heads/deadbeef`.

For safety, `west update` uses `git checkout --detach` to check out a detached `HEAD` at the manifest revision for each updated project, leaving behind any branches which were already checked out. This is typically a safe operation that will not modify any of your local branches.

However, if you had added some local commits onto a previously detached `HEAD` checked out by west, then git will warn you that you've left behind some commits which are no longer referred to by any

[1] West may fetch all refs from the Git server when given a SHA as a revision. This is because some Git servers have historically not allowed fetching SHAs directly.

branch. These may be garbage-collected and lost at some point in the future. To avoid this if you have local commits in the project, make sure you have a local branch checked out before running `west update`.

If you would rather rebase any locally checked out branches instead, use the `-r` (`--rebase`) option.

If you would like `west update` to keep local branches checked out as long as they point to commits that are descendants of the new `manifest-rev`, use the `-k` (`--keep-descendants`) option.

Note: `west update --rebase` will fail in projects that have git conflicts between your branch and new commits brought in by the manifest. You should immediately resolve these conflicts as you usually do with `git`, or you can use `git -C <project_path> rebase --abort` to ignore incoming changes for the moment.

With a clean working tree, a plain `west update` never fails because it does not try to hold on to your commits and simply leaves them aside.

`west update --keep-descendants` offers an intermediate option that never fails either but does not treat all projects the same:

- in projects where your branch diverged from the incoming commits, it does not even try to rebase and leaves your branches behind just like a plain `west update` does;
- in all other projects where no rebase or merge is needed it keeps your branches in place.

One-time project group manipulation:

The `--group-filter` option can be used to change which project groups are enabled or disabled for the duration of a single `west update` command. See *Project Groups and Active Projects* for details on the project group feature.

The `west update` command behaves as if the `--group-filter` option's value were appended to the `manifest.group-filter` *configuration option*.

For example, running `west update --group-filter=+foo,-bar` would behave the same way as if you had temporarily appended the string `"+foo,-bar"` to the value of `manifest.group-filter`, run `west update`, then restored `manifest.group-filter` to its original value.

Note that using the syntax `--group-filter=VALUE` instead of `--group-filter VALUE` avoids issues parsing command line options if you just want to disable a single group, e.g. `--group-filter=-bar`.

Submodule update procedure:

If a project in the manifest has a `submodules` key, the submodules are updated as follows, depending on the value of the `submodules` key.

If the project has `submodules: true`, west first synchronizes the project's submodules with:

```
git submodule sync --recursive
```

West then runs one of the following in the project repository, depending on whether you run `west update` with the `--rebase` option or without it:

```
# without --rebase, e.g. "west update":
git submodule update --init --checkout --recursive

# with --rebase, e.g. "west update --rebase":
git submodule update --init --rebase --recursive
```

Otherwise, the project has `submodules: <list-of-submodules>`. In this case, west synchronizes the project's submodules with:

```
git submodule sync --recursive -- <submodule-path>
```

Then it updates each submodule in the list as follows, depending on whether you run `west update` with the `--rebase` option or without it:

```
# without --rebase, e.g. "west update":
git submodule update --init --checkout --recursive <submodule-path>

# with --rebase, e.g. "west update --rebase":
git submodule update --init --rebase --recursive <submodule-path>
```

The `git submodule sync` commands are skipped if the `update.sync-submodules` *Configuration* option is false.

Other project commands

West has a few more commands for managing the projects in the workspace, which are summarized here. Run `west <command> -h` for detailed help.

- `west list`: print a line of information about each project in the manifest, according to a format string
- `west manifest`: manage the manifest file. See *Manifest Command*.
- `west diff`: run `git diff` in local project repositories
- `west status`: run `git status` in local project repositories
- `west forall`: run an arbitrary command in local project repositories

Other built-in commands

Finally, here is a summary of other built-in commands.

- `west config`: get or set *configuration options*
- `west topdir`: print the top level directory of the west workspace
- `west help`: get help about a command, or print information about all commands in the workspace, including *Extensions*

8.20.6 Workspaces

This page describes the *west workspace* concept introduced in *Basics* in more detail.

The `manifest-rev` branch

West creates and controls a Git branch named `manifest-rev` in each project. This branch points to the revision that the manifest file specified for the project at the time *west update* was last run. Other workspace management commands may use `manifest-rev` as a reference point for the upstream revision as of this latest update. Among other purposes, the `manifest-rev` branch allows the manifest file to use SHAs as project revisions.

Although `manifest-rev` is a normal Git branch, west will recreate and/or reset it on the next update. For this reason, it is **dangerous** to check it out or otherwise modify it yourself. For instance, any commits you manually add to this branch may be lost the next time you run `west update`. Instead, check out a local branch with another name, and either rebase it on top of a new `manifest-rev`, or merge `manifest-rev` into it.

Note: West does not create a `manifest-rev` branch in the manifest repository, since west does not manage the manifest repository's branches or revisions.

The `refs/west/*` Git refs

West also reserves all Git refs that begin with `refs/west/` (such as `refs/west/foo`) for itself in local project repositories. Unlike `manifest-rev`, these refs are not regular branches. West's behavior here is an implementation detail; users should not rely on these refs' existence or behavior.

Private repositories

You can use west to fetch from private repositories. There is nothing west-specific about this.

The `west update` command essentially runs `git fetch YOUR_PROJECT_URL` when a project's `manifest-rev` branch must be updated to a newly fetched commit. It's up to your environment to make sure the fetch succeeds.

You can either enter the password manually or use any of the credential helpers built in to Git. Since Git has credential storage built in, there is no need for a west-specific feature.

The following sections cover common cases for running `west update` without having to enter your password, as well as how to troubleshoot issues.

Fetching via HTTPS On Windows when fetching from GitHub, recent versions of Git prompt you for your GitHub password in a graphical window once, then store it for future use (in a default installation). Passwordless fetching from GitHub should therefore work "out of the box" on Windows after you have done it once.

In general, you can store your credentials on disk using the "store" git credential helper. See the git-credential-store manual page for details.

To use this helper for all the repositories in your workspace, run:

```
west forall -c "git config credential.helper store"
```

To use this helper on just the projects `foo` and `bar`, run:

```
west forall -c "git config credential.helper store" foo bar
```

To use this helper by default on your computer, run:

```
git config --global credential.helper store
```

On GitHub, you can set up a personal access token to use in place of your account password. (This may be required if your account has two-factor authentication enabled, and may be preferable to storing your account password in plain text even if two-factor authentication is disabled.)

If you don't want to store any credentials on the file system, you can store them in memory temporarily using git-credential-cache instead.

Fetching via SSH If your SSH key has no password, fetching should just work. If it does have a password, you can avoid entering it manually every time using ssh-agent.

On GitHub, see Connecting to GitHub with SSH for details on configuration and key creation.

Project locations

Projects can be located anywhere inside the workspace, but they may not "escape" it.

In other words, project repositories need not be located in subdirectories of the manifest repository or as immediate subdirectories of the topdir. However, projects must have paths inside the workspace.

You may replace a project's repository directory within the workspace with a symbolic link to elsewhere on your computer, but west will not do this for you.

Topologies supported

The following are example source code topologies supported by west.

- T1: star topology, zephyr is the manifest repository
- T2: star topology, a Zephyr application is the manifest repository
- T3: forest topology, freestanding manifest repository

T1: Star topology, zephyr is the manifest repository

- The zephyr repository acts as the central repository and specifies its *Modules (External projects)* in its `west.yml`
- Analogy with existing mechanisms: Git submodules with zephyr as the super-project

This is the default. See *Workspace concepts* for how mainline Zephyr is an example of this topology.

T2: Star topology, application is the manifest repository

- Useful for those focused on a single application
- A repository containing a Zephyr application acts as the central repository and names other projects required to build it in its `west.yml`. This includes the zephyr repository and any modules.
- Analogy with existing mechanisms: Git submodules with the application as the super-project, zephyr and other projects as submodules

A workspace using this topology looks like this:

```
west-workspace/
│
├── application/          # .git/      │
│   ├── CMakeLists.txt                 │
│   ├── prj.conf                       │  never modified by west
│   ├── src/                           │
│   │   └── main.c                     │
│   └── west.yml          # main manifest with optional import(s) and override(s)
│                                      │
├── modules/                           │
│   └── lib/                          │
│       └── tinycbor/     # .git/ project from either the main manifest or some import.
│
└── zephyr/               # .git/ project
    └── west.yml          # This can be partially imported with lower precedence or␣
↪ignored.
                          # Only the 'manifest-rev' version can be imported.
```

Here is an example `application/west.yml` which uses *Manifest Imports*, available since west 0.7, to import Zephyr v2.5.0 and its modules into the application manifest file:

```
# Example T2 west.yml, using manifest imports.
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.5.0
      import: true
  self:
    path: application
```

You can still selectively "override" individual Zephyr modules if you use `import:` in this way; see *Example 1.3: Downstream of a Zephyr release, with module fork* for an example.

Another way to do the same thing is to copy/paste `zephyr/west.yml` to `application/west.yml`, adding an entry for the zephyr project itself, like this:

```
# Equivalent to the above, but with manually maintained Zephyr modules.
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  defaults:
    remote: zephyrproject-rtos
  projects:
    - name: zephyr
      revision: v2.5.0
      west-commands: scripts/west-commands.yml
    - name: net-tools
      revision: some-sha-goes-here
      path: tools/net-tools
    # ... other Zephyr modules go here ...
  self:
    path: application
```

(The `west-commands` is there for *Building, Flashing and Debugging* and other Zephyr-specific *Extensions*. It's not necessary when using `import`.)

The main advantage to using `import` is not having to track the revisions of imported projects separately. In the above example, using `import` means Zephyr's *module* versions are automatically determined from the `zephyr/west.yml` revision, instead of having to be copy/pasted (and maintained) on their own.

T3: Forest topology

- Useful for those supporting multiple independent applications or downstream distributions with no "central" repository
- A dedicated manifest repository which contains no Zephyr source code, and specifies a list of projects all at the same "level"
- Analogy with existing mechanisms: Google repo-based source distribution

A workspace using this topology looks like this:

```
west-workspace/
├── app1/               # .git/ project
│   ├── CMakeLists.txt
│   ├── prj.conf
```

(continues on next page)

(continued from previous page)

```
│   └── src/
│       └── main.c
├── app2/               # .git/ project
│   ├── CMakeLists.txt
│   ├── prj.conf
│   └── src/
│       └── main.c
├── manifest-repo/      # .git/ never modified by west
│   └── west.yml        # main manifest with optional import(s) and override(s)
├── modules/
│   └── lib/
│       └── tinycbor/   # .git/ project from either the main manifest or
│                       #       from some import
│
└── zephyr/             # .git/ project
    └── west.yml        # This can be partially imported with lower precedence or␣
↪ignored.
                        # Only the 'manifest-rev' version can be imported.
```

Here is an example T3 `manifest-repo/west.yml` which uses *Manifest Imports*, available since west 0.7, to import Zephyr v2.5.0 and its modules, then add the `app1` and `app2` projects:

```
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
    - name: your-git-server
      url-base: https://git.example.com/your-company
  defaults:
    remote: your-git-server
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.5.0
      import: true
    - name: app1
      revision: SOME_SHA_OR_BRANCH_OR_TAG
    - name: app2
      revision: ANOTHER_SHA_OR_BRANCH_OR_TAG
  self:
    path: manifest-repo
```

You can also do this "by hand" by copy/pasting `zephyr/west.yml` as shown *above* for the T2 topology, with the same caveats.

8.20.7 West Manifests

This page contains detailed information about west's multiple repository model, manifest files, and the `west manifest` command. For API documentation on the `west.manifest` module, see west-apis-manifest. For a more general introduction and command overview, see *Basics*.

Multiple Repository Model

West's view of the repositories in a west workspace, and their history, looks like the following figure (though some parts of this example are specific to upstream Zephyr's use of west):

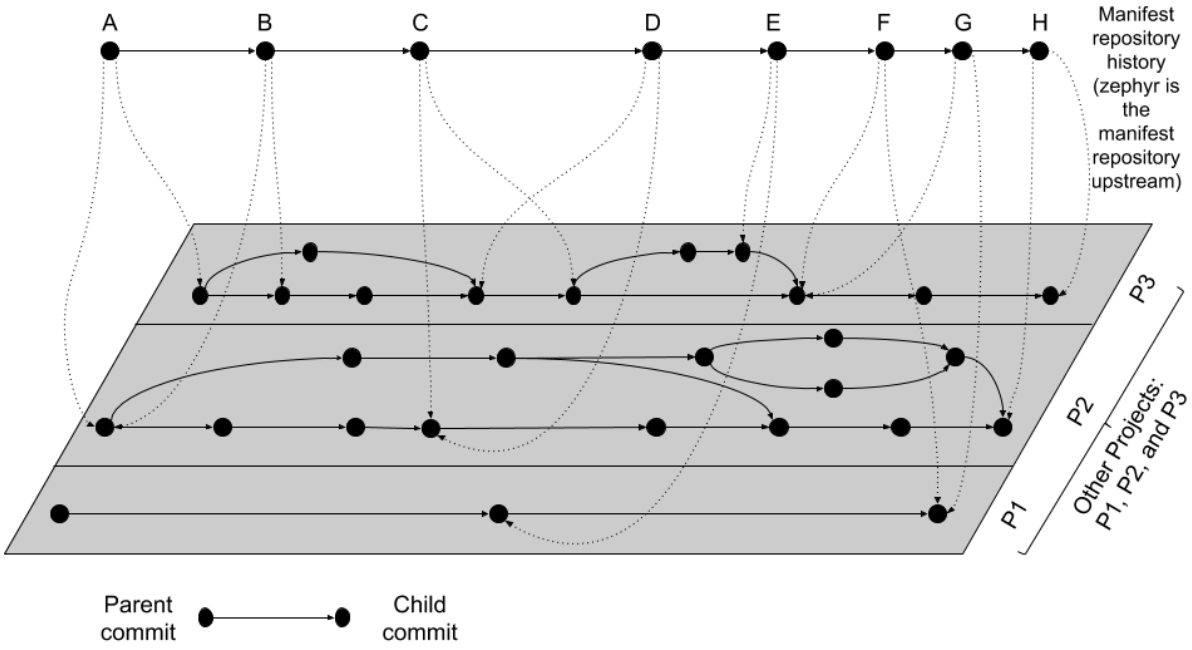


Fig. 13: West multi-repo history

The history of the manifest repository is the line of Git commits which is "floating" on top of the gray plane. Parent commits point to child commits using solid arrows. The plane below contains the Git commit history of the repositories in the workspace, with each project repository boxed in by a rectangle. Parent/child commit relationships in each repository are also shown with solid arrows.

The commits in the manifest repository (again, for upstream Zephyr this is the zephyr repository itself) each have a manifest file. The manifest file in each commit specifies the corresponding commits which it expects in each of the project repositories. This relationship is shown using dotted line arrows in the diagram. Each dotted line arrow points from a commit in the manifest repository to a corresponding commit in a project repository.

Notice the following important details:

- Projects can be added (like `P1` between manifest repository commits `D` and `E`) and removed (`P2` between the same manifest repository commits)
- Project and manifest repository histories don't have to move forwards or backwards together:
  - `P2` stays the same from `A` $\rightarrow$ `B`, as do `P1` and `P3` from `F` $\rightarrow$ `G`.
  - `P3` moves forward from `A` $\rightarrow$ `B`.
  - `P3` moves backward from `C` $\rightarrow$ `D`.

  One use for moving backward in project history is to "revert" a regression by going back to a revision before it was introduced.
- Project repository commits can be "skipped": `P3` moves forward multiple commits in its history from `B` $\rightarrow$ `C`.
- In the above diagram, no project repository has two revisions "at the same time": every manifest file refers to exactly one commit in the projects it cares about. This can be relaxed by using a branch name as a manifest revision, at the cost of being able to bisect manifest repository history.

Manifest Files

West manifests are YAML files. Manifests have a top-level `manifest` section with some subsections, like this:

```
manifest:
  remotes:
    # short names for project URLs
  projects:
    # a list of projects managed by west
  defaults:
    # default project attributes
  self:
    # configuration related to the manifest repository itself,
    # i.e. the repository containing west.yml
  version: "<schema-version>"
  group-filter:
    # a list of project groups to enable or disable
```

In YAML terms, the manifest file contains a mapping, with a `manifest` key. Any other keys and their contents are ignored (west v0.5 also required a `west` key, but this is ignored starting with v0.6).

The manifest contains subsections, like `defaults`, `remotes`, `projects`, and `self`. In YAML terms, the value of the `manifest` key is also a mapping, with these "subsections" as keys. As of west v0.10, all of these "subsection" keys are optional.

The `projects` value is a list of repositories managed by west and associated metadata. We'll discuss it soon, but first we will describe the `remotes` section, which can be used to save typing in the `projects` list.

Remotes The `remotes` subsection contains a sequence which specifies the base URLs where projects can be fetched from.

Each `remotes` element has a name and a "URL base". These are used to form the complete Git fetch URL for each project. A project's fetch URL can be set by appending a project-specific path onto a remote URL base. (As we'll see below, projects can also specify their complete fetch URLs.)

For example:

```
manifest:
  # ...
  remotes:
    - name: remote1
      url-base: https://git.example.com/base1
    - name: remote2
      url-base: https://git.example.com/base2
```

The `remotes` keys and their usage are in the following table.

Table 16: remotes keys

Key	Description
`name`	Mandatory; a unique name for the remote.
`url-base`	A prefix that is prepended to the fetch URL for each project with this remote.

Above, two remotes are given, with names `remote1` and `remote2`. Their URL bases are respectively `https://git.example.com/base1` and `https://git.example.com/base2`. You can use SSH URL bases as well; for example, you might use `git@example.com:base1` if `remote1` supported Git over SSH as well. Anything acceptable to Git will work.

Projects The `projects` subsection contains a sequence describing the project repositories in the west workspace. Every project has a unique name. You can specify what Git remote URLs to use when cloning and fetching the projects, what revisions to track, and where the project should be stored on the local file system.

Here is an example. We'll assume the `remotes` given above.

```
manifest:
  # [... same remotes as above...]
  projects:
    - name: proj1
      remote: remote1
      path: extra/project-1
    - name: proj2
      repo-path: my-path
      remote: remote2
      revision: v1.3
    - name: proj3
      url: https://github.com/user/project-three
      revision: abcde413a111
```

In this manifest:

- `proj1` has remote `remote1`, so its Git fetch URL is `https://git.example.com/base1/proj1`. The remote `url-base` is appended with a `/` and the project `name` to form the URL.

  Locally, this project will be cloned at path `extra/project-1` relative to the west workspace's root directory, since it has an explicit `path` attribute with this value.

  Since the project has no `revision` specified, `master` is used by default. The current tip of this branch will be fetched and checked out as a detached `HEAD` when west next updates this project.

- `proj2` has a `remote` and a `repo-path`, so its fetch URL is `https://git.example.com/base2/my-path`. The `repo-path` attribute, if present, overrides the default `name` when forming the fetch URL.

  Since the project has no `path` attribute, its `name` is used by default. It will be cloned into a directory named `proj2`. The commit pointed to by the `v1.3` tag will be checked out when west updates the project.

- `proj3` has an explicit `url`, so it will be fetched from `https://github.com/user/project-three`.

  Its local path defaults to its name, `proj3`. Commit `abcde413a111` will be checked out when it is next updated.

The available project keys and their usage are in the following table. Sometimes we'll refer to the `defaults` subsection; it will be described next.

Table 17: projects elements keys

Key(s)	Description
`name`	Mandatory; a unique name for the project. The name cannot be one of the reserved values "west" or "manifest". The name must be unique in the manifest file.
`remote`, `url`	Mandatory (one of the two, but not both). If the project has a `remote`, that remote's `url-base` will be combined with the project's `name` (or `repo-path`, if it has one) to form the fetch URL instead. If the project has a `url`, that's the complete fetch URL for the remote Git repository. If the project has neither, the `defaults` section must specify a `remote`, which will be used as the the project's remote. Otherwise, the manifest is invalid.
`repo-path`	Optional. If given, this is concatenated on to the remote's `url-base` instead of the project's `name` to form its fetch URL. Projects may not have both `url` and `repo-path` attributes.
`revision`	Optional. The Git revision that `west update` should check out. This will be checked out as a detached HEAD by default, to avoid conflicting with local branch names. If not given, the `revision` value from the `defaults` subsection will be used if present. A project revision can be a branch, tag, or SHA. The default `revision` is `master` if not otherwise specified.
`path`	Optional. Relative path specifying where to clone the repository locally, relative to the top directory in the west workspace. If missing, the project's `name` is used as a directory name.
`clone-depth`	Optional. If given, a positive integer which creates a shallow history in the cloned repository limited to the given number of commits. This can only be used if the `revision` is a branch or tag.
`west-commands`	Optional. If given, a relative path to a YAML file within the project which describes additional west commands provided by that project. This file is named `west-commands.yml` by convention. See *Extensions* for details.
`import`	Optional. If `true`, imports projects from manifest files in the given repository into the current manifest. See *Manifest Imports* for details.
`groups`	Optional, a list of groups the project belongs to. See *Project Groups and Active Projects* for details.
`submodules`	Optional. You can use this to make `west update` also update Git submodules defined by the project. See *Git Submodules in Projects* for details.

Defaults The `defaults` subsection can provide default values for project attributes. In particular, the default remote name and revision can be specified here. Another way to write the same manifest we have been describing so far using `defaults` is:

```
manifest:
  defaults:
    remote: remote1
    revision: v1.3

  remotes:
    - name: remote1
      url-base: https://git.example.com/base1
    - name: remote2
      url-base: https://git.example.com/base2

  projects:
    - name: proj1
      path: extra/project-1
      revision: master
    - name: proj2
      repo-path: my-path
```

(continues on next page)

(continued from previous page)

```
      remote: remote2
    - name: proj3
      url: https://github.com/user/project-three
      revision: abcde413a111
```

The available `defaults` keys and their usage are in the following table.

Table 18: defaults keys

Key	Description
`remote`	Optional. This will be used for a project's `remote` if it does not have a `url` or `remote` key set.
`revision`	Optional. This will be used for a project's `revision` if it does not have one set. If not given, the default is `master`.

Self The `self` subsection can be used to control the manifest repository itself.

As an example, let's consider this snippet from the zephyr repository's `west.yml`:

```
manifest:
  # ...
  self:
    path: zephyr
    west-commands: scripts/west-commands.yml
```

This ensures that the zephyr repository is cloned into path `zephyr`, though as explained above that would have happened anyway if cloning from the default manifest URL, `https://github.com/zephyrproject-rtos/zephyr`. Since the zephyr repository does contain extension commands, its `self` entry declares the location of the corresponding `west-commands.yml` relative to the repository root.

The available `self` keys and their usage are in the following table.

Table 19: self keys

Key	Description
`path`	Optional. The path `west init` should clone the manifest repository into, relative to the west workspace topdir. If not given, the basename of the path component in the manifest repository URL will be used by default. For example, if the URL is `https://git.example.com/project-repo`, the manifest repository would be cloned to the directory `project-repo`.
`west-commands`	Optional. This is analogous to the same key in a project sequence element.
`import`	Optional. This is also analogous to the `projects` key, but allows importing projects from other files in the manifest repository. See *Manifest Imports*.

Version The `version` subsection can be used to mark the lowest version of the manifest file schema that can parse this file's data:

```
manifest:
  version: "0.10"
  # marks that this file uses version 0.10 of the west manifest
  # file format.
```

The pykwalify schema `manifest-schema.yml` in the west source code repository is used to validate the manifest section. The current manifest `version` is 0.10, which is supported by west version v0.10.x.

The `version` value may be `"0.7"`, `"0.8"`, `"0.9"`, or `"0.10"`. West v0.10.x can load manifests with any of these `version` values, while west v0.9.x can only load versions up to `"0.9"`, and so on.

West halts with an error if you ask it to load a manifest file written in a version it cannot handle.

Quoting the `version` value as shown above forces the YAML parser to treat it as a string. Without quotes, `0.10` in YAML is just the floating point value `0.1`. You can omit the quotes if the value is the same when cast to string, but it's best to include them. Always use quotes if you're not sure.

Group-filter See *Project Groups and Active Projects*.

Project Groups and Active Projects

You can use the `groups` and `group-filter` keys briefly described *above* to place projects into groups, and filter which groups are enabled. These keys appear in the manifest like this:

```
manifest:
  projects:
    - name: some-project
      groups: ...
  group-filter: ...
```

You can enable or disable project groups using `group-filter`. Projects whose groups are all disabled are *inactive*; west essentially ignores inactive projects unless explicitly requested not to.

The next section introduces project groups; the following sections describe *Enabled and Disabled Project Groups* and *Active and Inactive Projects*. There are some basic examples in *Project Group Examples*.

Finally, *Group Filters and Imports* provides a simplified overview of how `group-filter` interacts with the *Manifest Imports* feature.

Project Groups Inside `manifest: projects:`, you can add a project to one or more groups. The `groups` key is a list of group names. Group names are strings.

For example, in this manifest fragment:

```
manifest:
  projects:
    - name: project-1
      groups:
        - groupA
    - name: project-2
      groups:
        - groupB
        - groupC
    - name: project-3
```

The projects are in these groups:

- `project-1`: one group, named `groupA`
- `project-2`: two groups, named `groupB` and `groupC`
- `project-3`: no groups

Project group names must not contain commas (,), colons (:), or whitespace.

Group names must not begin with a dash (-) or the plus sign (+), but they may contain these characters elsewhere in their names. For example, `foo-bar` and `foo+bar` are valid groups, but `-foobar` and `+foobar` are not.

Group names are otherwise arbitrary strings. Group names are case sensitive.

As a restriction, no project may use both `import:` and `groups:`. (This avoids some edge cases whose semantics are difficult to specify.)

Enabled and Disabled Project Groups All project groups are enabled by default. You can enable or disable groups in both your manifest file and *Configuration*.

Within a manifest file, `manifest: group-filter:` is a YAML list of groups to enable and disable.

To enable a group, prefix its name with a plus sign (+). For example, `groupA` is enabled in this manifest fragment:

```
manifest:
  group-filter: [+groupA]
```

Although this is redundant for groups that are already enabled by default, it can be used to override settings in an imported manifest file. See *Group Filters and Imports* for more information.

To disable a group, prefix its name with a dash (-). For example, `groupA` and `groupB` are disabled in this manifest fragment:

```
manifest:
  group-filter: [-groupA,-groupB]
```

Note: Since `group-filter` is a YAML list, you could have written this fragment as follows:

```
manifest:
  group-filter:
    - -groupA
    - -groupB
```

However, this syntax is harder to read and therefore discouraged.

In addition to the manifest file, you can control which groups are enabled and disabled using the `manifest.group-filter` configuration option. This option is a comma-separated list of groups to enable and/or disable.

To enable a group, add its name to the list prefixed with `+`. To disable a group, add its name prefixed with `-`. For example, setting `manifest.group-filter` to `+groupA,-groupB` enables `groupA`, and disables `groupB`.

The value of the configuration option overrides any data in the manifest file. You can think of this as if the `manifest.group-filter` configuration option is appended to the `manifest: group-filter:` list from YAML, with "last entry wins" semantics.

Active and Inactive Projects All projects are *active* by default. Projects with no groups are always active. A project is *inactive* if all of its groups are disabled. This is the only way to make a project inactive.

Most west commands that operate on projects will ignore inactive projects by default. For example, *west update* when run without arguments will not update inactive projects. As another example, running `west list` without arguments will not print information for inactive projects.

Project Group Examples This section contains example situations involving project groups and active projects. The examples use both `manifest: group-filter:` YAML lists and `manifest.group-filter` configuration lists, to show how they work together.

Note that the `defaults` and `remotes` data in the following manifests isn't relevant except to make the examples complete and self-contained.

Example 1: no disabled groups The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB
    - name: baz

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

No groups are disabled, because all groups are enabled by default. Therefore, all three projects (`foo`, `bar`, and `baz`) are active. Note that there is no way to make project `baz` inactive, since it has no groups.

Example 2: Disabling one group via manifest The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB

  group-filter: [-groupA]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

Since `groupA` is disabled, project `foo` is inactive. Project `bar` is active, because `groupB` is enabled.

Example 3: Disabling multiple groups via manifest The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
```

(continues on next page)

(continued from previous page)

```
        - groupA
        - groupB

  group-filter: [-groupA,-groupB]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

Both `foo` and `bar` are inactive, because all of their groups are disabled.

Example 4: Disabling a group via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `-groupA` (you can ensure this by running `west config manifest.group-filter -- -groupA`; the extra `--` is required so the argument parser does not treat `-groupA` as a command line option `-g` with value `roupA`).

Project `foo` is inactive because `groupA` has been disabled by the `manifest.group-filter` configuration option. Project `bar` is active because `groupB` is enabled.

Example 5: Overriding a disabled group via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  group-filter: [-groupA]
```

(continues on next page)

(continued from previous page)

```
  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `+groupA` (you can ensure this by running `west config manifest.group-filter +groupA`).

In this case, `groupA` is enabled: the `manifest.group-filter` configuration option has higher precedence than the `manifest: group-filter: [-groupA]` content in the manifest file.

Therefore, projects `foo` and `bar` are both active.

Example 6: Overriding multiple disabled groups via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  group-filter: [-groupA,-groupB]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `+groupA,+groupB` (you can ensure this by running `west config manifest.group-filter "+groupA,+groupB"`).

In this case, both `groupA` and `groupB` are enabled, because the configuration value overrides the manifest file for both groups.

Therefore, projects `foo` and `bar` are both active.

Example 7: Disabling multiple groups via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  defaults:
```

(continues on next page)

(continued from previous page)

```
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `-groupA,-groupB` (you can ensure this by running `west config manifest.group-filter -- "-groupA,-groupB"`).

In this case, both `groupA` and `groupB` are disabled.

Therefore, projects `foo` and `bar` are both inactive.

Group Filters and Imports This section provides a simplified description of how the `manifest: group-filter:` value behaves when combined with *Manifest Imports*. For complete details, see *Manifest Import Details*.

Warning: The below semantics apply to west v0.10.0 and later. West v0.9.x semantics are different, and combining `group-filter` with `import` in west v0.9.x is discouraged.

In short:

- if you only import one manifest, any groups it disables in its `group-filter` are also disabled in your manifest
- you can override this in your manifest file's `manifest: group-filter:` value, your workspace's `manifest.group-filter` configuration option, or both

Here are some examples.

Example 1: no overrides You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

Only `child` and `project-2` are active in the resolved manifest.

The `unstable` group is disabled in `child/west.yml`, and that is not overridden in `parent/west.yml`. Therefore, the final `group-filter` for the resolved manifest is `[-unstable]`.

Since `project-1` and `project-3` are in the `unstable` group and are not in any other group, they are inactive.

Example 2: overriding an imported `group-filter` via manifest You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  group-filter: [+unstable,-optional]
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
      groups:
        - optional
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

Only the `child`, `project-1`, and `project-3` projects are active.

The `[-unstable]` group filter in `child/west.yml` is overridden in `parent/west.yml`, so the `unstable` group is enabled. Since `project-1` and `project-3` are in the `unstable` group, they are active.

The same `parent/west.yml` file disables the `optional` group, so `project-2` is inactive.

The final group filter specified by `parent/west.yml` is `[+unstable,-optional]`.

Example 3: overriding an imported `group-filter` via configuration You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
      groups:
        - optional
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

If you run:

```
west config manifest.group-filter +unstable,-optional
```

Then only the `child`, `project-1`, and `project-3` projects are active.

The `-unstable` group filter in `child/west.yml` is overridden in the `manifest.group-filter` configuration option, so the `unstable` group is enabled. Since `project-1` and `project-3` are in the `unstable` group, they are active.

The same configuration option disables the `optional` group, so `project-2` is inactive.

The final group filter specified by `parent/west.yml` and the `manifest.group-filter` configuration option is `[+unstable,-optional]`.

Git Submodules in Projects

You can use the `submodules` keys briefly described *above* to force `west update` to also handle any Git submodules configured in project's git repository. The `submodules` key can appear inside `projects`, like this:

```
manifest:
  projects:
    - name: some-project
      submodules: ...
```

The `submodules` key can be a boolean or a list of mappings. We'll describe these in order.

Option 1: Boolean This is the easiest way to use `submodules`.

If `submodules` is `true` as a `projects` attribute, `west update` will recursively update the project's Git submodules whenever it updates the project itself. If it's `false` or missing, it has no effect.

For example, let's say you have a source code repository `foo`, which has some submodules, and you want `west update` to keep all of them them in sync, along with another project named `bar` in the same workspace.

You can do that with this manifest file:

```
manifest:
  projects:
    - name: foo
      submodules: true
    - name: bar
```

Here, `west update` will initialize and update all submodules in `foo`. If `bar` has any submodules, they are ignored, because `bar` does not have a `submodules` value.

Option 2: List of mappings The `submodules` key may be a list of mappings, one list element for each desired submodule. Each submodule listed is updated recursively. You can still track and update unlisted submodules with `git` commands manually; present or not they will be completely ignored by `west`.

The `path` key must match exactly the path of one submodule relative to its parent west project, as shown in the output of `git submodule status`. The `name` key is optional and not used by west for now; it's not passed to `git submodule` commands either. The `name` key was briefly mandatory in west version 0.9.0, but was made optional in 0.9.1.

For example, let's say you have a source code repository `foo`, which has many submodules, and you want `west update` to keep some but not all of them in sync, along with another project named `bar` in the same workspace.

You can do that with this manifest file:

```
manifest:
  projects:
    - name: foo
      submodules:
        - path: path/to/foo-first-sub
        - name: foo-second-sub
          path: path/to/foo-second-sub
    - name: bar
```

Here, `west update` will recursively initialize and update just the submodules in `foo` with paths `path/to/foo-first-sub` and `path/to/foo-second-sub`. Any submodules in `bar` are still ignored.

Manifest Imports

You can use the `import` key briefly described above to include projects from other manifest files in your `west.yml`. This key can be either a `project` or `self` section attribute:

```
manifest:
  projects:
    - name: some-project
      import: ...
  self:
    import: ...
```

You can use a "self: import:" to load additional files from the repository containing your `west.yml`. You can use a "project: ... import:" to load additional files defined in that project's Git history.

West resolves the final manifest from individual manifest files in this order:

1. imported files in `self`
2. your `west.yml` file
3. imported files in `projects`

During resolution, west ignores projects which have already been defined in other files. For example, a project named `foo` in your `west.yml` makes west ignore other projects named `foo` imported from your `projects` list.

The `import` key can be a boolean, path, mapping, or sequence. We'll describe these in order, using examples:

- *Boolean*

A more *formal description* of how this works is last, after the examples.

Troubleshooting Note If you're using this feature and find west's behavior confusing, try *resolving your manifest* to see the final results after imports are done.

Option 1: Boolean This is the easiest way to use `import`.

If `import` is `true` as a `projects` attribute, west imports projects from the `west.yml` file in that project's root directory. If it's `false` or missing, it has no effect. For example, this manifest would import `west.yml` from the `p1` git repository at revision `v1.0`:

```
manifest:
  # ...
  projects:
    - name: p1
      revision: v1.0
      import: true    # Import west.yml from p1's v1.0 git tag
    - name: p2
      import: false   # Nothing is imported from p2.
    - name: p3        # Nothing is imported from p3 either.
```

It's an error to set `import` to either `true` or `false` inside `self`, like this:

```
manifest:
  # ...
  self:
    import: true  # Error
```

Example 1.1: Downstream of a Zephyr release You have a source code repository you want to use with Zephyr v1.14.1 LTS. You want to maintain the whole thing using west. You don't want to modify any of the mainline repositories.

In other words, the west workspace you want looks like this:

```
my-downstream/
├── .west/                      # west directory
├── zephyr/                     # mainline zephyr repository
│   └── west.yml                # the v1.14.1 version of this file is imported
├── modules/                    # modules from mainline zephyr
│   ├── hal/
│   └── [...other directories..]
├── [ ... other projects ...]  # other mainline repositories
└── my-repo/                    # your downstream repository
    ├── west.yml                # main manifest importing zephyr/west.yml v1.14.1
    └── [...other files..]
```

You can do this with the following `my-repo/west.yml`:

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v1.14.1
      import: true
```

You can then create the workspace on your computer like this, assuming `my-repo` is hosted at `https://git.example.com/my-repo`:

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

After `west init`, `my-downstream/my-repo` will be cloned.

After `west update`, all of the projects defined in the `zephyr` repository's `west.yml` at revision `v1.14.1` will be cloned into `my-downstream` as well.

You can add and commit any code to `my-repo` you please at this point, including your own Zephyr applications, drivers, etc. See *Application Development*.

Example 1.2: "Rolling release" Zephyr downstream This is similar to *Example 1.1: Downstream of a Zephyr release*, except we'll use `revision: main` for the zephyr repository:

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: main
      import: true
```

You can create the workspace in the same way:

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

This time, whenever you run `west update`, the special *manifest-rev* branch in the `zephyr` repository will be updated to point at a newly fetched `main` branch tip from the URL https://github.com/zephyrproject-rtos/zephyr.

The contents of `zephyr/west.yml` at the new `manifest-rev` will then be used to import projects from Zephyr. This lets you stay up to date with the latest changes in the Zephyr project. The cost is that running `west update` will not produce reproducible results, since the remote `main` branch can change every time you run it.

It's also important to understand that west **ignores your working tree's** `zephyr/west.yml` entirely when resolving imports. West always uses the contents of imported manifests as they were committed to the latest `manifest-rev` when importing from a project.

You can only import manifest from the file system if they are in your manifest repository's working tree. See *Example 2.2: Downstream with directory of manifest files* for an example.

Example 1.3: Downstream of a Zephyr release, with module fork This manifest is similar to the one in *Example 1.1: Downstream of a Zephyr release*, except it:

- is a downstream of Zephyr 2.0
- includes a downstream fork of the `modules/hal/nordic` *module* which was included in that release

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
    - name: my-remote
      url-base: https://git.example.com
  projects:
    - name: hal_nordic          # higher precedence
      remote: my-remote
      revision: my-sha
      path: modules/hal/nordic
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.0.0
      import: true              # imported projects have lower precedence

# subset of zephyr/west.yml contents at v2.0.0:
manifest:
  defaults:
    remote: zephyrproject-rtos
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
  # ...
  - name: hal_nordic            # lower precedence, values ignored
    path: modules/hal/nordic
    revision: another-sha
```

With this manifest file, the project named `hal_nordic`:

- is cloned from `https://git.example.com/hal_nordic` instead of `https://github.com/zephyrproject-rtos/hal_nordic`.
- is updated to commit `my-sha` by `west update`, instead of the mainline commit `another-sha`

In other words, when your top-level manifest defines a project, like `hal_nordic`, west will ignore any other definition it finds later on while resolving imports.

This does mean you have to copy the `path: modules/hal/nordic` value into `my-repo/west.yml` when defining `hal_nordic` there. The value from `zephyr/west.yml` is ignored entirely. See *Resolving Manifests* for troubleshooting advice if this gets confusing in practice.

When you run `west update`, west will:

- update zephyr's `manifest-rev` to point at the `v2.0.0` tag
- import `zephyr/west.yml` at that `manifest-rev`
- locally check out the `v2.0.0` revisions for all zephyr projects except `hal_nordic`
- update `hal_nordic` to `my-sha` instead of `another-sha`

Option 2: Relative path The `import` value can also be a relative path to a manifest file or a directory containing manifest files. The path is relative to the root directory of the `projects` or `self` repository the `import` key appears in.

Here is an example:

```
manifest:
  projects:
    - name: project-1
      revision: v1.0
      import: west.yml
    - name: project-2
      revision: main
      import: p2-manifests
  self:
    import: submanifests
```

This will import the following:

- the contents of `project-1/west.yml` at `manifest-rev`, which points at tag `v1.0` after running `west update`
- any YAML files in the directory tree `project-2/p2-manifests` at the latest commit in the `main` branch, as fetched by `west update`, sorted by file name
- YAML files in `submanifests` in your manifest repository, as they appear on your file system, sorted by file name

Notice how `projects` imports get data from Git using `manifest-rev`, while `self` imports get data from your file system. This is because as usual, west leaves version control for your manifest repository up to you.

Example 2.1: Downstream of a Zephyr release with explicit path This is an explicit way to write an equivalent manifest to the one in *Example 1.1: Downstream of a Zephyr release*.

```
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v1.14.1
      import: west.yml
```

The setting `import: west.yml` means to use the file `west.yml` inside the `zephyr` project. This example is contrived, but shows the idea.

This can be useful in practice when the name of the manifest file you want to import is not `west.yml`.

Example 2.2: Downstream with directory of manifest files Your Zephyr downstream has a lot of additional repositories. So many, in fact, that you want to split them up into multiple manifest files, but keep track of them all in a single manifest repository, like this:

```
my-repo/
├── submanifests
│   ├── 01-libraries.yml
│   ├── 02-vendor-hals.yml
│   └── 03-applications.yml
└── west.yml
```

You want to add all the files in `my-repo/submanifests` to the main manifest file, `my-repo/west.yml`, in addition to projects in `zephyr/west.yml`. You want to track the latest development code in the Zephyr repository's `main` branch instead of using a fixed revision.

Here's how:

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: main
      import: true
  self:
    import: submanifests
```

Manifest files are imported in this order during resolution:

1. `my-repo/submanifests/01-libraries.yml`
2. `my-repo/submanifests/02-vendor-hals.yml`
3. `my-repo/submanifests/03-applications.yml`
4. `my-repo/west.yml`
5. `zephyr/west.yml`

Note: The `.yml` file names are prefixed with numbers in this example to make sure they are imported in the specified order.

You can pick arbitrary names. West sorts files in a directory by name before importing.

Notice how the manifests in `submanifests` are imported *before* `my-repo/west.yml` and `zephyr/west.yml`. In general, an `import` in the `self` section is processed before the manifest files in `projects` and the main manifest file.

This means projects defined in `my-repo/submanifests` take highest precedence. For example, if `01-libraries.yml` defines `hal_nordic`, the project by the same name in `zephyr/west.yml` is simply ignored. As usual, see *Resolving Manifests* for troubleshooting advice.

This may seem strange, but it allows you to redefine projects "after the fact", as we'll see in the next example.

Example 2.3: Continuous Integration overrides Your continuous integration system needs to fetch and test multiple repositories in your west workspace from a developer's forks instead of your mainline development trees, to see if the changes all work well together.

Starting with *Example 2.2: Downstream with directory of manifest files*, the CI scripts add a file `00-ci.yml` in `my-repo/submanifests`, with these contents:

```
# my-repo/submanifests/00-ci.yml:
manifest:
  projects:
    - name: a-vendor-hal
      url: https://github.com/a-developer/hal
      revision: a-pull-request-branch
    - name: an-application
      url: https://github.com/a-developer/application
      revision: another-pull-request-branch
```

The CI scripts run `west update` after generating this file in `my-repo/submanifests`. The projects defined in `00-ci.yml` have higher precedence than other definitions in `my-repo/submanifests`, because the name `00-ci.yml` comes before the other file names.

Thus, `west update` always checks out the developer's branches in the projects named `a-vendor-hal` and `an-application`, even if those same projects are also defined elsewhere.

Option 3: Mapping The `import` key can also contain a mapping with the following keys:

- `file`: Optional. The name of the manifest file or directory to import. This defaults to `west.yml` if not present.
- `name-allowlist`: Optional. If present, a name or sequence of project names to include.
- `path-allowlist`: Optional. If present, a path or sequence of project paths to match against. This is a shell-style globbing pattern, currently implemented with pathlib. Note that this means case sensitivity is platform specific.
- `name-blocklist`: Optional. Like `name-allowlist`, but contains project names to exclude rather than include.
- `path-blocklist`: Optional. Like `path-allowlist`, but contains project paths to exclude rather than include.
- `path-prefix`: Optional (new in v0.8.0). If given, this will be prepended to the project's path in the workspace, as well as the paths of any imported projects. This can be used to place these projects in a subdirectory of the workspace.

Allowlists override blocklists if both are given. For example, if a project is blocked by path, then allowed by name, it will still be imported.

Example 3.1: Downstream with name allowlist Here is a pair of manifest files, representing a mainline and a downstream. The downstream doesn't want to use all the mainline projects, however. We'll assume the mainline `west.yml` is hosted at `https://git.example.com/mainline/manifest`.

```
# mainline west.yml:
manifest:
  projects:
    - name: mainline-app                # included
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib
      path: libraries/lib
      url: https://git.example.com/mainline/lib
    - name: lib2                        # included
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2
```

(continues on next page)

(continued from previous page)

```
# downstream west.yml:
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
      import:
        name-allowlist:
          - mainline-app
          - lib2
    - name: downstream-app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

An equivalent manifest in a single file would be:

```
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
    - name: downstream-app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
    - name: mainline-app                    # imported
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib2                            # imported
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2
```

If an allowlist had not been used, the `lib` project from the mainline manifest would have been imported.

Example 3.2: Downstream with path allowlist Here is an example showing how to allowlist mainline's libraries only, using `path-allowlist`.

```
# mainline west.yml:
manifest:
  projects:
    - name: app
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib
      path: libraries/lib                   # included
      url: https://git.example.com/mainline/lib
    - name: lib2
      path: libraries/lib2                  # included
      url: https://git.example.com/mainline/lib2

# downstream west.yml:
manifest:
  projects:
    - name: mainline
```

(continues on next page)

(continued from previous page)

```
      url: https://git.example.com/mainline/manifest
      import:
        path-allowlist: libraries/*
    - name: app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

An equivalent manifest in a single file would be:

```
manifest:
  projects:
    - name: lib                          # imported
      path: libraries/lib
      url: https://git.example.com/mainline/lib
    - name: lib2                         # imported
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2
    - name: mainline
      url: https://git.example.com/mainline/manifest
    - name: app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

Example 3.3: Downstream with path blocklist Here's an example showing how to block all vendor HALs from mainline by common path prefix in the workspace, add your own version for the chip you're targeting, and keep everything else.

```
# mainline west.yml:
manifest:
  defaults:
    remote: mainline
  remotes:
    - name: mainline
      url-base: https://git.example.com/mainline
  projects:
    - name: app
    - name: lib
      path: libraries/lib
    - name: lib2
      path: libraries/lib2
    - name: hal_foo
      path: modules/hals/foo     # excluded
    - name: hal_bar
      path: modules/hals/bar     # excluded
    - name: hal_baz
      path: modules/hals/baz     # excluded

# downstream west.yml:
manifest:
  projects:
    - name: mainline
```

(continues on next page)

(continued from previous page)

```
      url: https://git.example.com/mainline/manifest
      import:
        path-blocklist: modules/hals/*
    - name: hal_foo
      path: modules/hals/foo
      url: https://git.example.com/downstream/hal_foo
```

An equivalent manifest in a single file would be:

```
manifest:
  defaults:
    remote: mainline
  remotes:
    - name: mainline
      url-base: https://git.example.com/mainline
  projects:
    - name: app                  # imported
    - name: lib                  # imported
      path: libraries/lib
    - name: lib2                 # imported
      path: libraries/lib2
    - name: mainline
      repo-path: https://git.example.com/mainline/manifest
    - name: hal_foo
      path: modules/hals/foo
      url: https://git.example.com/downstream/hal_foo
```

Example 3.4: Import into a subdirectory You want to import a manifest and its projects, placing everything into a subdirectory of your west workspace.

For example, suppose you want to import this manifest from project `foo`, adding this project and its projects `bar` and `baz` to your workspace:

```
# foo/west.yml:
manifest:
  defaults:
    remote: example
  remotes:
    - name: example
      url-base: https://git.example.com
  projects:
    - name: bar
    - name: baz
```

Instead of importing these into the top level workspace, you want to place all three project repositories in an `external-code` subdirectory, like this:

```
workspace/
└── external-code/
    ├── foo/
    ├── bar/
    └── baz/
```

You can do this using this manifest:

```
manifest:
  projects:
    - name: foo
      url: https://git.example.com/foo
      import:
        path-prefix: external-code
```

An equivalent manifest in a single file would be:

```
# foo/west.yml:
manifest:
  defaults:
    remote: example
  remotes:
    - name: example
      url-base: https://git.example.com
  projects:
    - name: foo
      path: external-code/foo
    - name: bar
      path: external-code/bar
    - name: baz
      path: external-code/baz
```

Option 4: Sequence The `import` key can also contain a sequence of files, directories, and mappings.

Example 4.1: Downstream with sequence of manifest files This example manifest is equivalent to the manifest in *Example 2.2: Downstream with directory of manifest files*, with a sequence of explicitly named files.

```
# my-repo/west.yml:
manifest:
  projects:
    - name: zephyr
      url: https://github.com/zephyrproject-rtos/zephyr
      import: west.yml
  self:
    import:
      - submanifests/01-libraries.yml
      - submanifests/02-vendor-hals.yml
      - submanifests/03-applications.yml
```

Example 4.2: Import order illustration This more complicated example shows the order that west imports manifest files:

```
# my-repo/west.yml
manifest:
  # ...
  projects:
    - name: my-library
    - name: my-app
    - name: zephyr
      import: true
    - name: another-manifest-repo
```

(continues on next page)

(continued from previous page)

```
    import: submanifests
  self:
    import:
      - submanifests/libraries.yml
      - submanifests/vendor-hals.yml
      - submanifests/applications.yml
  defaults:
    remote: my-remote
```

For this example, west resolves imports in this order:

1. the listed files in `my-repo/submanifests` are first, in the order they occur (e.g. `libraries.yml` comes before `applications.yml`, since this is a sequence of files), since the `self: import:` is always imported first
2. `my-repo/west.yml` is next (with projects `my-library` etc. as long as they weren't already defined somewhere in `submanifests`)
3. `zephyr/west.yml` is after that, since that's the first `import` key in the `projects` list in `my-repo/west.yml`
4. files in `another-manifest-repo/submanifests` are last (sorted by file name), since that's the final project `import`

Manifest Import Details This section describes how west resolves a manifest file that uses `import` a bit more formally.

Overview The `import` key can appear in a west manifest's `projects` and `self` sections. The general case looks like this:

```
# Top-level manifest file.
manifest:
  projects:
    - name: foo
      import: import-1
    - name: bar
      import: import-2
    # ...
    - name: baz
      import: import-N
  self:
    import: self-import
```

Import keys are optional. If any of `import-1, ..., import-N` are missing, west will not import additional manifest data from that project. If `self-import` is missing, no additional files in the manifest repository (beyond the top-level file) are imported.

The ultimate outcomes of resolving manifest imports are:

- a `projects` list, which is produced by combining the `projects` defined in the top-level file with those defined in imported files
- a set of extension commands, which are drawn from the the `west-commands` keys in in the top-level file and any imported files
- a `group-filter` list, which is produced by combining the top-level and any imported filters

Importing is done in this order:

1. Manifests from `self-import` are imported first.

2. The top-level manifest file's definitions are handled next.
3. Manifests from `import-1`, ..., `import-N`, are imported in that order.

When an individual `import` key refers to multiple manifest files, they are processed in this order:

- If the value is a relative path naming a directory (or a map whose `file` is a directory), the manifest files it contains are processed in lexicographic order – i.e., sorted by file name.
- If the value is a sequence, its elements are recursively imported in the order they appear.

This process recurses if necessary. E.g., if `import-1` produces a manifest file that contains an `import` key, it is resolved recursively using the same rules before its contents are processed further.

Projects This section describes how the final `projects` list is created.

Projects are identified by name. If the same name occurs in multiple manifests, the first definition is used, and subsequent definitions are ignored. For example, if `import-1` contains a project named `bar`, that is ignored, because the top-level `west.yml` has already defined a project by that name.

The contents of files named by `import-1` through `import-N` are imported from Git at the latest `manifest-rev` revisions in their projects. These revisions can be updated to the values `rev-1` through `rev-N` by running `west update`. If any `manifest-rev` reference is missing or out of date, `west update` also fetches project data from the remote fetch URL and updates the reference.

Also note that all imported manifests, from the root manifest to the repository which defines a project `P`, must be up to date in order for west to update `P` itself. For example, this means `west update P` would update `manifest-rev` in the `baz` project if `baz/west.yml` defines `P`, as well as updating the `manifest-rev` branch in the local git clone of `P`. Confusingly, updating `baz` may result in the removal of `P` from `baz/west.yml`, which "should" cause `west update P` to fail with an unrecognized project!

For this reason, it's not possible to run `west update P` if `P` is defined in an imported manifest; you must update this project along with all the others with a plain `west update`.

By default, west won't fetch any project data over the network if a project's revision is a SHA or tag which is already available locally, so updating the extra projects shouldn't take too much time unless it's really needed. See the documentation for the *update.fetch* configuration option for more information.

Extensions All extension commands defined using `west-commands` keys discovered while handling imports are available in the resolved manifest.

If an imported manifest file has a `west-commands:` definition in its `self:` section, the extension commands defined there are added to the set of available extensions at the time the manifest is imported. They will thus take precedence over any extension commands with the same names added later on.

Group filters The resolved manifest has a `group-filter` value which is the result of concatenating the `group-filter` values in the top-level manifest and any imported manifests.

Manifest files which appear earlier in the import order have higher precedence and are therefore concatenated later into the final `group-filter`.

In other words, let:

- the submanifest resolved from `self-import` have group filter `self-filter`
- the top-level manifest file have group filter `top-filter`
- the submanifests resolved from `import-1` through `import-N` have group filters `filter-1` through `filter-N` respectively

The final resolved `group-filter` value is then `filter1 + filter-2 + ... + filter-N + top-filter + self-filter`, where `+` here refers to list concatenation.

Important: The order that filters appear in the above list matters.

The last filter element in the final concatenated list "wins" and determines if the group is enabled or disabled.

For example, in `[-foo] + [+foo]`, group `foo` is *enabled*. However, in `[+foo] + [-foo]`, group `foo` is *disabled*.

For simplicity, west and this documentation may elide concatenated group filter elements which are redundant using these rules. For example, `[+foo] + [-foo]` could be written more simply as `[-foo]`, for the reasons given above. As another example, `[-foo] + [+foo]` could be written as the empty list `[]`, since all groups are enabled by default.

Manifest Command

The `west manifest` command can be used to manipulate manifest files. It takes an action, and action-specific arguments.

The following sections describe each action and provides a basic signature for simple uses. Run `west manifest --help` for full details on all options.

Resolving Manifests The `--resolve` action outputs a single manifest file equivalent to your current manifest and all its *imported manifests*:

```
west manifest --resolve [-o outfile]
```

The main use for this action is to see the "final" manifest contents after performing any `imports`.

To print detailed information about each imported manifest file and how projects are handled during manifest resolution, set the maximum verbosity level using `-v`:

```
west -v manifest --resolve
```

Freezing Manifests The `--freeze` action outputs a frozen manifest:

```
west manifest --freeze [-o outfile]
```

A "frozen" manifest is a manifest file where every project's revision is a SHA. You can use `--freeze` to produce a frozen manifest that's equivalent to your current manifest file. The `-o` option specifies an output file; if not given, standard output is used.

Validating Manifests The `--validate` action either succeeds if the current manifest file is valid, or fails with an error:

```
west manifest --validate
```

The error message can help diagnose errors.

Get the manifest path The `--path` action prints the path to the top level manifest file:

```
west manifest --path
```

The output is something like `/path/to/workspace/west.yml`. The path format depends on your operating system.

8.20.8 Configuration

This page documents west's configuration file system, the `west config` command, and configuration options used by built-in commands. For API documentation on the `west.configuration` module, see west-apis-configuration.

West Configuration Files

West's configuration file syntax is INI-like; here is an example file:

```
[manifest]
path = zephyr

[zephyr]
base = zephyr
```

Above, the `manifest` section has option `path` set to `zephyr`. Another way to say the same thing is that `manifest.path` is `zephyr` in this file.

There are three types of configuration file:

1. **System**: Settings in this file affect west's behavior for every user logged in to the computer. Its location depends on the platform:
   - Linux: `/etc/westconfig`
   - macOS: `/usr/local/etc/westconfig`
   - Windows: `%PROGRAMDATA%\west\config`
2. **Global** (per user): Settings in this file affect how west behaves when run by a particular user on the computer.
   - All platforms: the default is `.westconfig` in the user's home directory.
   - Linux note: if the environment variable `XDG_CONFIG_HOME` is set, then `$XDG_CONFIG_HOME/west/config` is used.
   - Windows note: the following environment variables are tested to find the home directory: `%HOME%`, then `%USERPROFILE%`, then a combination of `%HOMEDRIVE%` and `%HOMEPATH%`.
3. **Local**: Settings in this file affect west's behavior for the current west workspace. The file is `.west/config`, relative to the workspace's root directory.

A setting in a file which appears lower down on this list overrides an earlier setting. For example, if `color.ui` is `true` in the system's configuration file, but `false` in the workspace's, then the final value is `false`. Similarly, settings in the user configuration file override system settings, and so on.

west config

The built-in `config` command can be used to get and set configuration values. You can pass `west config` the options `--system`, `--global`, or `--local` to specify which configuration file to use. Only one of these can be used at a time. If none is given, then writes default to `--local`, and reads show the final value after applying overrides.

Some examples for common uses follow; run `west config -h` for detailed help, and see *Built-in Configuration Options* for more details on built-in options.

To set `manifest.path` to `some-other-manifest`:

```
west config manifest.path some-other-manifest
```

Doing the above means that commands like `west update` will look for the west manifest inside the `some-other-manifest` directory (relative to the workspace root directory) instead of the directory given to `west init`, so be careful!

To read `zephyr.base`, the value which will be used as `ZEPHYR_BASE` if it is unset in the calling environment (also relative to the workspace root):

```
west config zephyr.base
```

You can switch to another zephyr repository without changing `manifest.path` – and thus the behavior of commands like `west update` – using:

```
west config zephyr.base some-other-zephyr
```

This can be useful if you use commands like `git worktree` to create your own zephyr directories, and want commands like `west build` to use them instead of the zephyr repository specified in the manifest. (You can go back to using the directory in the upstream manifest by running `west config zephyr.base zephyr`.)

To set `color.ui` to `false` in the global (user-wide) configuration file, so that west will no longer print colored output for that user when run in any workspace:

```
west config --global color.ui false
```

To undo the above change:

```
west config --global color.ui true
```

Built-in Configuration Options

The following table documents configuration options supported by west's built-in commands. Configuration options supported by Zephyr's extension commands are documented in the pages for those commands.

Option	Description
`color.ui`	Boolean. If `true` (the default), then west output is colorized when stdout is a terminal.
`commands.allow_extensions`	Boolean, default `true`, disables *Extensions* if `false`
`manifest.file`	String, default `west.yml`. Relative path from the manifest repository root directory to the manifest file used by `west init` and other commands which parse the manifest.
`manifest.group-filter`	String, default empty. A comma-separated list of project groups to enable and disable within the workspace. Prefix enabled groups with `+` and disabled groups with `-`. For example, the value `"+foo,-bar"` enables group `foo` and disables `bar`. See *Project Groups and Active Projects*.
`manifest.path`	String, relative path from the west workspace root directory to the manifest repository used by `west update` and other commands which parse the manifest. Set locally by `west init`.
`update.fetch`	String, one of `"smart"` (the default behavior starting in v0.6.1) or `"always"` (the previous behavior). If set to `"smart"`, the *west update* command will skip fetching from project remotes when those projects' revisions in the manifest file are SHAs or tags which are already available locally. The `"always"` behavior is to unconditionally fetch from the remote.
`update.name-cache`	String. If non-empty, `west update` will use its value as the `--name-cache` option's value if not given on the command line.
`update.narrow`	Boolean. If `true`, `west update` behaves as if `--narrow` was given on the command line. The default is `false`.
`update.path-cache`	String. If non-empty, `west update` will use its value as the `--path-cache` option's value if not given on the command line.
`update.sync-submodules`	Boolean. If `true` (the default), *west update* will synchronize Git submodules before updating them.
`zephyr.base`	String, default value to set for the `ZEPHYR_BASE` environment variable while the west command is running. By default, this is set to the path to the manifest project with path `zephyr` (if there is one) during `west init`. If the variable is already set, then this setting is ignored unless `zephyr.base-prefer` is `"configfile"`.
`zephyr.base-prefer`	String, one the values `"env"` and `"configfile"`. If set to `"env"` (the default), setting `ZEPHYR_BASE` in the calling environment overrides the value of the `zephyr.base` configuration option. If set to `"configfile"`, the configuration option wins instead.

8.20.9 Extensions

West is "pluggable": you can add your own commands to west without editing its source code. These are called **west extension commands**, or just "extensions" for short. Extensions show up in the `west --help` output in a special section for the project which defines them. This page provides general information on west extension commands, and has a tutorial for writing your own.

Some commands you can run when using west with Zephyr, like the ones used to *build, flash, and debug* and the *ones described here* , are extensions. That's why help for them shows up like this in `west --help`:

```
commands from project at "zephyr":
  completion:           display shell completion scripts
  boards:               display information about supported boards
  build:                compile a Zephyr application
  sign:                 sign a Zephyr binary for bootloader chain-loading
  flash:                flash and run a binary on a board
  debug:                flash and interactively debug a Zephyr application
```

(continues on next page)

(continued from previous page)

```
  debugserver:          connect to board and launch a debug server
  attach:               interactively debug a board
```

See `zephyr/scripts/west-commands.yml` and the `zephyr/scripts/west_commands` directory for the implementation details.

Disabling Extension Commands

To disable support for extension commands, set the `commands.allow_extensions` *configuration* option to `false`. To set this globally for whenever you run west, use:

```
west config --global commands.allow_extensions false
```

If you want to, you can then re-enable them in a particular west workspace with:

```
west config --local commands.allow_extensions true
```

Note that the files containing extension commands are not imported by west unless the commands are explicitly run. See below for details.

Adding a West Extension

There are three steps to adding your own extension:

1. Write the code implementing the command.
2. Add information about it to a `west-commands.yml` file.
3. Make sure the `west-commands.yml` file is referenced in the west manifest.

Note that west ignores extension commands whose names are the same as a built-in command.

Step 1: Implement Your Command Create a Python file to contain your command implementation (see the "Meta > Requires" information on the west PyPI page for details on the currently supported versions of Python). You can put it in anywhere in any project tracked by your west manifest, or the manifest repository itself. This file must contain a subclass of the `west.commands.WestCommand` class; this class will be instantiated and used when your extension is run.

Here is a basic skeleton you can use to get started. It contains a subclass of `WestCommand`, with implementations for all the abstract methods. For more details on the west APIs you can use, see west-apis.

```
'''my_west_extension.py

Basic example of a west extension.'''

from textwrap import dedent             # just for nicer code indentation

from west.commands import WestCommand   # your extension must subclass this
from west import log                    # use this for user output

class MyCommand(WestCommand):

    def __init__(self):
        super().__init__(
            'my-command-name',  # gets stored as self.name
            'one-line help for what my-command-name does',  # self.help
            # self.description:
```

(continues on next page)

(continued from previous page)

```
            dedent('''
            A multi-line description of my-command.

            You can split this up into multiple paragraphs and they'll get
            reflowed for you. You can also pass
            formatter_class=argparse.RawDescriptionHelpFormatter when calling
            parser_adder.add_parser() below if you want to keep your line
            endings.'''))

    def do_add_parser(self, parser_adder):
        # This is a bit of boilerplate, which allows you full control over the
        # type of argparse handling you want. The "parser_adder" argument is
        # the return value of an argparse.ArgumentParser.add_subparsers() call.
        parser = parser_adder.add_parser(self.name,
                                         help=self.help,
                                         description=self.description)

        # Add some example options using the standard argparse module API.
        parser.add_argument('-o', '--optional', help='an optional argument')
        parser.add_argument('required', help='a required argument')

        return parser           # gets stored as self.parser

    def do_run(self, args, unknown_args):
        # This gets called when the user runs the command, e.g.:
        #
        #   $ west my-command-name -o FOO BAR
        #   --optional is FOO
        #   required is BAR
        log.inf('--optional is', args.optional)
        log.inf('required is', args.required)
```

You can ignore the second argument to `do_run()` (`unknown_args` above), as `WestCommand` will reject unknown arguments by default. If you want to be passed a list of unknown arguments instead, add `accepts_unknown_args=True` to the `super().__init__()` arguments.

Step 2: Add or Update Your `west-commands.yml` You now need to add a `west-commands.yml` file to your project which describes your extension to west.

Here is an example for the above class definition, assuming it's in `my_west_extension.py` at the project root directory:

```
west-commands:
  - file: my_west_extension.py
    commands:
      - name: my-command-name
        class: MyCommand
        help: one-line help for what my-command-name does
```

The top level of this YAML file is a map with a `west-commands` key. The key's value is a sequence of "command descriptors". Each command descriptor gives the location of a file implementing west extensions, along with the names of those extensions, and optionally the names of the classes which define them (if not given, the `class` value defaults to the same thing as `name`).

Some information in this file is redundant with definitions in the Python code. This is because west won't import `my_west_extension.py` until the user runs `west my-command-name`, since:

- It allows users to run `west update` with a manifest from an untrusted source, then use other west

commands without your code being imported along the way. Since importing a Python module is shell-equivalent, this provides some peace of mind.

- It's a small optimization, since your code will only be imported if it is needed.

So, unless your command is explicitly run, west will just load the `west-commands.yml` file to get the basic information it needs to display information about your extension to the user in `west --help` output, etc.

If you have multiple extensions, or want to split your extensions across multiple files, your `west-commands.yml` will look something like this:

```
west-commands:
  - file: my_west_extension.py
    commands:
      - name: my-command-name
        class: MyCommand
        help: one-line help for what my-command-name does
  - file: another_file.py
    commands:
      - name: command2
        help: another cool west extension
      - name: a-third-command
        class: ThirdCommand
        help: a third command in the same file as command2
```

Above:

- `my_west_extension.py` defines extension `my-command-name` with class `MyCommand`
- `another_file.py` defines two extensions:
  1. `command2` with class `command2`
  2. `a-third-command` with class `ThirdCommand`

See the file `west-commands-schema.yml` in the west repository for a schema describing the contents of a `west-comands.yml`.

Step 3: Update Your Manifest Finally, you need to specify the location of the `west-commands.yml` you just edited in your west manifest. If your extension is in a project, add it like this:

```
manifest:
   # [... other contents ...]

   projects:
     - name: your-project
       west-commands: path/to/west-commands.yml
     # [... other projects ...]
```

Where `path/to/west-commands.yml` is relative to the root of the project. Note that the name `west-commands.yml`, while encouraged, is just a convention; you can name the file something else if you need to.

Alternatively, if your extension is in the manifest repository, just do the same thing in the manifest's `self` section, like this:

```
manifest:
  # [... other contents ...]

  self:
    west-commands: path/to/west-commands.yml
```

That's it; you can now run `west my-command-name`. Your command's name, help, and the project which contains its code will now also show up in the `west --help` output. If you share the updated repositories with others, they'll be able to use it, too.

8.20.10 Building, Flashing and Debugging

Zephyr provides several *west extension commands* for building, flashing, and interacting with Zephyr programs running on a board: `build`, `flash`, `debug`, `debugserver` and `attach`.

For information on adding board support for the flashing and debugging commands, see *Flash and debug support* in the board porting guide.

Building: `west build`

Tip: Run `west build -h` for a quick overview.

The `build` command helps you build Zephyr applications from source. You can use *west config* to configure its behavior.

Its default behavior tries to "do what you mean":

- If there is a Zephyr build directory named `build` in your current working directory, it is incrementally re-compiled. The same is true if you run `west build` from a Zephyr build directory.
- Otherwise, if you run `west build` from a Zephyr application's source directory and no build directory is found, a new one is created and the application is compiled in it.

Basics The easiest way to use `west build` is to go to an application's root directory (i.e. the folder containing the application's `CMakeLists.txt`) and then run:

```
west build -b <BOARD>
```

Where `<BOARD>` is the name of the board you want to build for. This is exactly the same name you would supply to CMake if you were to invoke it with: `cmake -DBOARD=<BOARD>`.

Tip: You can use the *west boards* command to list all supported boards.

A build directory named `build` will be created, and the application will be compiled there after `west build` runs CMake to create a build system in that directory. If `west build` finds an existing build directory, the application is incrementally re-compiled there without re-running CMake. You can force CMake to run again with `--cmake`.

You don't need to use the `--board` option if you've already got an existing build directory; `west build` can figure out the board from the CMake cache. For new builds, the `--board` option, `BOARD` environment variable, or `build.board` configuration option are checked (in that order).

Examples Here are some `west build` usage examples, grouped by area.

Forcing CMake to Run Again To force a CMake re-run, use the `--cmake` (or `--c`) option:

```
west build -c
```

Setting a Default Board To configure `west build` to build for the `reel_board` by default:

```
west config build.board reel_board
```

(You can use any other board supported by Zephyr here; it doesn't have to be `reel_board`.)

Setting Source and Build Directories To set the application source directory explicitly, give its path as a positional argument:

```
west build -b <BOARD> path/to/source/directory
```

To set the build directory explicitly, use `--build-dir` (or `-d`):

```
west build -b <BOARD> --build-dir path/to/build/directory
```

To change the default build directory from `build`, use the `build.dir-fmt` configuration option. This lets you name build directories using format strings, like this:

```
west config build.dir-fmt "build/{board}/{app}"
```

With the above, running `west build -b reel_board samples/hello_world` will use build directory `build/reel_board/hello_world`. See *Configuration Options* for more details on this option.

Setting the Build System Target To specify the build system target to run, use `--target` (or `-t`).

For example, on host platforms with QEMU, you can use the `run` target to build and run the hello_world sample for the emulated qemu_x86 board in one command:

```
west build -b qemu_x86 -t run samples/hello_world
```

As another example, to use `-t` to list all build system targets:

```
west build -t help
```

As a final example, to use `-t` to run the `pristine` target, which deletes all the files in the build directory:

```
west build -t pristine
```

Pristine Builds A *pristine* build directory is essentially a new build directory. All byproducts from previous builds have been removed.

To force `west build` make the build directory pristine before re-running CMake to generate a build system, use the `--pristine=always` (or `-p=always`) option.

Giving `--pristine` or `-p` without a value has the same effect as giving it the value `always`. For example, these commands are equivalent:

```
west build -p -b reel_board samples/hello_world
west build -p=always -b reel_board samples/hello_world
```

By default, `west build` applies a heuristic to detect if the build directory needs to be made pristine. This is the same as using `--pristine=auto`.

Tip: You can run `west config build.pristine always` to always do a pristine build, or `west config build.pristine never` to disable the heuristic. See the `west build` *Configuration Options* for details.

Verbose Builds To print the CMake and compiler commands run by `west build`, use the global west verbosity option, `-v`:

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

One-Time CMake Arguments To pass additional arguments to the CMake invocation performed by `west build`, pass them after a `--` at the end of the command line.

Important: Passing additional CMake arguments like this forces `west build` to re-run CMake, even if a build system has already been generated.

After using `--` once to generate the build directory, use `west build -d <build-dir>` on subsequent runs to do incremental builds.

For example, to use the Unix Makefiles CMake generator instead of Ninja (which `west build` uses by default), run:

```
west build -b reel_board -- -G'Unix Makefiles'
```

To use Unix Makefiles and set CMAKE_VERBOSE_MAKEFILE to `ON`:

```
west build -b reel_board -- -G'Unix Makefiles' -DCMAKE_VERBOSE_MAKEFILE=ON
```

Notice how the `--` only appears once, even though multiple CMake arguments are given. All command-line arguments to `west build` after a `--` are passed to CMake.

To set *DTC_OVERLAY_FILE* to `enable-modem.overlay`, using that file as a *devicetree overlay*:

```
west build -b reel_board -- -DDTC_OVERLAY_FILE=enable-modem.overlay
```

To merge the `file.conf` Kconfig fragment into your build's `.config`:

```
west build -- -DOVERLAY_CONFIG=file.conf
```

Permanent CMake Arguments The previous section describes how to add CMake arguments for a single `west build` command. If you want to save CMake arguments for `west build` to use every time it generates a new build system instead, you should use the `build.cmake-args` configuration option. Whenever `west build` runs CMake to generate a build system, it splits this option's value according to shell rules and includes the results in the `cmake` command line.

Remember that, by default, `west build` **tries to avoid generating a new build system if one is present** in your build directory. Therefore, you need to either delete any existing build directories or do a *pristine build* after setting `build.cmake-args` to make sure it will take effect.

For example, to always enable `CMAKE_EXPORT_COMPILE_COMMANDS`, you can run:

```
west config build.cmake-args -- -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

(The extra `--` is used to force the rest of the command to be treated as a positional argument. Without it, *west config* would treat the `-DVAR=VAL` syntax as a use of its `-D` option.)

To enable `CMAKE_VERBOSE_MAKEFILE`, so CMake always produces a verbose build system:

```
west config build.cmake-args -- -DCMAKE_VERBOSE_MAKEFILE=ON
```

To save more than one argument in `build.cmake-args`, use a single string whose value can be split into distinct arguments (`west build` uses the Python function shlex.split() internally to split the value).

For example, to enable both `CMAKE_EXPORT_COMPILE_COMMANDS` and `CMAKE_VERBOSE_MAKEFILE`:

```
west config build.cmake-args -- "-DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DCMAKE_VERBOSE_
↪MAKEFILE=ON"
```

If you want to save your CMake arguments in a separate file instead, you can combine CMake's `-C <initial-cache>` option with `build.cmake-args`. For instance, another way to set the options used in the previous example is to create a file named `~/my-cache.cmake` with the following contents:

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON CACHE BOOL "")
set(CMAKE_VERBOSE_MAKEFILE ON CACHE BOOL "")
```

Then run:

```
west config build.cmake-args "-C ~/my-cache.cmake"
```

See the cmake(1) manual page and the set() command documentation for more details.

Build tool arguments Use `-o` to pass options to the underlying build tool.

This works with both `ninja` (*the default*) and `make` based build systems.

For example, to pass `-dexplain` to `ninja`:

```
west build -o=-dexplain
```

As another example, to pass `--keep-going` to `make`:

```
west build -o=--keep-going
```

Note that using `-o=--foo` instead of `-o --foo` is required to prevent `--foo` from being treated as a `west build` option.

Build parallelism By default, `ninja` uses all of your cores to build, while `make` uses only one. You can control this explicitly with the `-j` option supported by both tools.

For example, to build with 4 cores:

```
west build -o=-j4
```

The `-o` option is described further in the previous section.

Configuration Options You can *configure* `west build` using these options.

Option	Description
`build.board`	String. If given, this the board used by *west build* when `--board` is not given and `BOARD` is unset in the environment.
`build.board_warn`	Boolean, default `true`. If `false`, disables warnings when `west build` can't figure out the target board.
`build.cmake-args`	String. If present, the value will be split according to shell rules and passed to CMake whenever a new build system is generated. See *Permanent CMake Arguments*.
`build.dir-fmt`	String, default `build`. The build folder format string, used by west whenever it needs to create or locate a build folder. The currently available arguments are: • `board`: The board name • `source_dir`: The relative path from the current working directory to the source directory. If the current working directory is inside the source directory this will be set to an empty string. • `app`: The name of the source directory.
`build.generator`	String, default `Ninja`. The CMake Generator to use to create a build system. (To set a generator for a single build, see the *above example*)
`build.guess-dir`	String, instructs west whether to try to guess what build folder to use when `build.dir-fmt` is in use and not enough information is available to resolve the build folder name. Can take these values: • `never` (default): Never try to guess, bail out instead and require the user to provide a build folder with `-d`. • `runners`: Try to guess the folder when using any of the 'runner' commands. These are typically all commands that invoke an external tool, such as `flash` and `debug`.
`build.pristine`	String. Controls the way in which `west build` may clean the build folder before building. Can take the following values: • `never` (default): Never automatically make the build folder pristine. • `auto`: `west build` will automatically make the build folder pristine before building, if a build system is present and the build would fail otherwise (e.g. the user has specified a different board or application from the one previously used to make the build directory). • `always`: Always make the build folder pristine before building, if a build system is present.

Flashing: `west flash`

Tip: Run `west flash -h` for additional help.

Basics From a Zephyr build directory, re-build the binary and flash it to your board:

```
west flash
```

Without options, the behavior is the same as `ninja flash` (or `make flash`, etc.).

To specify the build directory, use `--build-dir` (or `-d`):

```
west flash --build-dir path/to/build/directory
```

If you don't specify the build directory, `west flash` searches for one in `build`, then the current working directory. If you set the `build.dir-fmt` configuration option (see *Setting Source and Build Directories*), `west flash` searches there instead of `build`.

Choosing a Runner If your board's Zephyr integration supports flashing with multiple programs, you can specify which one to use using the `--runner` (or `-r`) option. For example, if West flashes your board with `nrfjprog` by default, but it also supports JLink, you can override the default with:

```
west flash --runner jlink
```

You can override the default flash runner at build time by using the `BOARD_FLASH_RUNNER` CMake variable, and the debug runner with `BOARD_DEBUG_RUNNER`.

For example:

```
# Set the default runner to "jlink", overriding the board's
# usual default.
west build [...] -- -DBOARD_FLASH_RUNNER=jlink
```

See *One-Time CMake Arguments* and *Permanent CMake Arguments* for more information on setting CMake arguments.

See *Flash and debug runners* below for more information on the `runner` library used by West. The list of runners which support flashing can be obtained with `west flash -H`; if run from a build directory or with `--build-dir`, this will print additional information on available runners for your board.

Configuration Overrides The CMake cache contains default values West uses while flashing, such as where the board directory is on the file system, the path to the zephyr binaries to flash in several formats, and more. You can override any of this configuration at runtime with additional options.

For example, to override the HEX file containing the Zephyr image to flash (assuming your runner expects a HEX file), but keep other flash configuration at default values:

```
west flash --hex-file path/to/some/other.hex
```

The `west flash -h` output includes a complete list of overrides supported by all runners.

Runner-Specific Overrides Each runner may support additional options related to flashing. For example, some runners support an `--erase` flag, which mass-erases the flash storage on your board before flashing the Zephyr image.

To view all of the available options for the runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west flash --context
```

Important: Note the capital H in the short option name. This re-runs the build in order to ensure the information displayed is up to date!

When running West outside of a build directory, `west flash -H` just prints a list of runners. You can use `west flash -H -r <runner-name>` to print usage information for options supported by that runner.

For example, to print usage information about the `jlink` runner:

```
west flash -H -r jlink
```

Debugging: `west debug`, `west debugserver`

Tip: Run `west debug -h` or `west debugserver -h` for additional help.

Basics From a Zephyr build directory, to attach a debugger to your board and open up a debug console (e.g. a GDB session):

```
west debug
```

To attach a debugger to your board and open up a local network port you can connect a debugger to (e.g. an IDE debugger):

```
west debugserver
```

Without options, the behavior is the same as `ninja debug` and `ninja debugserver` (or `make debug`, etc.).

To specify the build directory, use `--build-dir` (or `-d`):

```
west debug --build-dir path/to/build/directory
west debugserver --build-dir path/to/build/directory
```

If you don't specify the build directory, these commands search for one in `build`, then the current working directory. If you set the `build.dir-fmt` configuration option (see *Setting Source and Build Directories*), `west debug` searches there instead of `build`.

Choosing a Runner If your board's Zephyr integration supports debugging with multiple programs, you can specify which one to use using the `--runner` (or `-r`) option. For example, if West debugs your board with `pyocd-gdbserver` by default, but it also supports JLink, you can override the default with:

```
west debug --runner jlink
west debugserver --runner jlink
```

See *Flash and debug runners* below for more information on the `runner` library used by West. The list of runners which support debugging can be obtained with `west debug -H`; if run from a build directory or with `--build-dir`, this will print additional information on available runners for your board.

Configuration Overrides The CMake cache contains default values West uses for debugging, such as where the board directory is on the file system, the path to the zephyr binaries containing symbol tables, and more. You can override any of this configuration at runtime with additional options.

For example, to override the ELF file containing the Zephyr binary and symbol tables (assuming your runner expects an ELF file), but keep other debug configuration at default values:

```
west debug --elf-file path/to/some/other.elf
west debugserver --elf-file path/to/some/other.elf
```

The `west debug -h` output includes a complete list of overrides supported by all runners.

Runner-Specific Overrides Each runner may support additional options related to debugging. For example, some runners support flags which allow you to set the network ports used by debug servers.

To view all of the available options for the runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west debug --context
```

(The command `west debugserver --context` will print the same output.)

Important: Note the capital H in the short option name. This re-runs the build in order to ensure the information displayed is up to date!

When running West outside of a build directory, `west debug -H` just prints a list of runners. You can use `west debug -H -r <runner-name>` to print usage information for options supported by that runner.

For example, to print usage information about the `jlink` runner:

```
west debug -H -r jlink
```

Flash and debug runners

The flash and debug commands use Python wrappers around various *Flash & Debug Host Tools*. These wrappers are all defined in a Python library at scripts/west_commands/runners. Each wrapper is called a *runner*. Runners can flash and/or debug Zephyr programs.

The central abstraction within this library is `ZephyrBinaryRunner`, an abstract class which represents runners. The set of available runners is determined by the imported subclasses of `ZephyrBinaryRunner`. `ZephyrBinaryRunner` is available in the `runners.core` module; individual runner implementations are in other submodules, such as `runners.nrfjprog`, `runners.openocd`, etc.

Hacking

This section documents the `runners.core` module used by the flash and debug commands. This is the core abstraction used to implement support for these features.

Warning: These APIs are provided for reference, but they are more "shared code" used to implement multiple extension commands than a stable API.

Developers can add support for new ways to flash and debug Zephyr programs by implementing additional runners. To get this support into upstream Zephyr, the runner should be added into a new or existing `runners` module, and imported from `runners/__init__.py`.

Note: The test cases in scripts/west_commands/tests add unit test coverage for the runners package and individual runner classes.

Please try to add tests when adding new runners. Note that if your changes break existing test cases, CI testing on upstream pull requests will fail.

Zephyr binary runner core interfaces

This provides the core ZephyrBinaryRunner class meant for public use, as well as some other helpers for concrete runner classes.

`class runners.core.BuildConfiguration(`*build_dir: str*`)`

This helper class provides access to build-time configuration.

Configuration options can be read as if the object were a dict, either object['CONFIG_FOO'] or object.get('CONFIG_FOO').

Kconfig configuration values are available (parsed from .config).

getboolean(*option*)

If a boolean option is explicitly set to y or n, returns its value. Otherwise, falls back to False.

exception runners.core.MissingProgram(*program*)

FileNotFoundError subclass for missing program dependencies.

No significant changes from the parent FileNotFoundError; this is useful for explicitly signaling that the file in question is a program that some class requires to proceed.

The filename attribute contains the missing program.

class runners.core.NetworkPortHelper

Helper class for dealing with local IP network ports.

get_unused_ports(*starting_from*)

Find unused network ports, starting at given values.

starting_from is an iterable of ports the caller would like to use.

The return value is an iterable of ports, in the same order, using the given values if they were unused, or the next sequentially available unused port otherwise.

Ports may be bound between this call's check and actual usage, so callers still need to handle errors involving returned ports.

class runners.core.RunnerCaps(*commands: Set[str] = {'attach', 'debug', 'debugserver', 'flash'}, flash_addr: bool = False, erase: bool = False*)

This class represents a runner class's capabilities.

Each capability is represented as an attribute with the same name. Flag attributes are True or False.

Available capabilities:

- commands: set of supported commands; default is {'flash', 'debug', 'debugserver', 'attach'}.
- flash_addr: whether the runner supports flashing to an arbitrary address. Default is False. If true, the runner must honor the –dt-flash option.
- erase: whether the runner supports an –erase option, which does a mass-erase of the entire addressable flash on the target before flashing. On multi-core SoCs, this may only erase portions of flash specific the actual target core. (This option can be useful for things like clearing out old settings values or other subsystem state that may affect the behavior of the zephyr image. It is also sometimes needed by SoCs which have flash-like areas that can't be sector erased by the underlying tool before flashing; UICR on nRF SoCs is one example.)

class runners.core.RunnerConfig(*build_dir: str, board_dir: str, elf_file: Optional[str], hex_file: Optional[str], bin_file: Optional[str], gdb: Optional[str] = None, openocd: Optional[str] = None, openocd_search: List[str] = []*)

Runner execution-time configuration.

This is a common object shared by all runners. Individual runners can register specific configuration options using their do_add_parser() hooks.

bin_file: Optional[str]

Alias for field number 4

board_dir: str

Alias for field number 1

build_dir: str

Alias for field number 0

elf_file: Optional[str]

Alias for field number 2

`gdb: Optional[str]`
Alias for field number 5

`hex_file: Optional[str]`
Alias for field number 3

`openocd: Optional[str]`
Alias for field number 6

`openocd_search: List[str]`
Alias for field number 7

`class runners.core.ZephyrBinaryRunner(`*cfg:* runners.core.RunnerConfig`)`

Abstract superclass for binary runners (flashers, debuggers).

Note: this class's API has changed relatively rarely since it as added, but it is not considered a stable Zephyr API, and may change without notice.

With some exceptions, boards supported by Zephyr must provide generic means to be flashed (have a Zephyr firmware binary permanently installed on the device for running) and debugged (have a breakpoint debugger and program loader on a host workstation attached to a running target).

This is supported by four top-level commands managed by the Zephyr build system:

- 'flash': flash a previously configured binary to the board, start execution on the target, then return.
- 'debug': connect to the board via a debugging protocol, program the flash, then drop the user into a debugger interface with symbol tables loaded from the current binary, and block until it exits.
- 'debugserver': connect via a board-specific debugging protocol, then reset and halt the target. Ensure the user is now able to connect to a debug server with symbol tables loaded from the binary.
- 'attach': connect to the board via a debugging protocol, then drop the user into a debugger interface with symbol tables loaded from the current binary, and block until it exits. Unlike 'debug', this command does not program the flash.

This class provides an API for these commands. Every subclass is called a 'runner' for short. Each runner has a name (like 'pyocd'), and declares commands it can handle (like 'flash'). Boards (like 'nrf52dk_nrf52832') declare which runner(s) are compatible with them to the Zephyr build system, along with information on how to configure the runner to work with the board.

The build system will then place enough information in the build directory to create and use runners with this class's create() method, which provides a command line argument parsing API. You can also create runners by instantiating subclasses directly.

In order to define your own runner, you need to:

1. Define a ZephyrBinaryRunner subclass, and implement its abstract methods. You may need to override capabilities().
2. Make sure the Python module defining your runner class is imported, e.g. by editing this package's __init__.py (otherwise, get_runners() won't work).
3. Give your runner's name to the Zephyr build system in your board's board.cmake.

Additional advice:

- If you need to import any non-standard-library modules, make sure to catch ImportError and defer complaints about it to a RuntimeError if one is missing. This avoids affecting users that don't require your runner, while still making it clear what went wrong to users that do require it that don't have the necessary modules installed.

- If you need to ask the user something (e.g. using input()), do it in your create() classmethod, not do_run(). That ensures your __init__() really has everything it needs to call do_run(), and also avoids calling input() when not instantiating within a command line application.
- Use self.logger to log messages using the standard library's logging API; your logger is named "runner.<your-runner-name()>"

For command-line invocation from the Zephyr build system, runners define their own argparse-based interface through the common add_parser() (and runner-specific do_add_parser() it delegates to), and provide a way to create instances of themselves from a RunnerConfig and parsed runner-specific arguments via create().

Runners use a variety of host tools and configuration values, the user interface to which is abstracted by this class. Each runner subclass should take any values it needs to execute one of these commands in its constructor. The actual command execution is handled in the run() method.

`classmethod add_parser(`*parser*`)`

Adds a sub-command parser for this runner.

The given object, parser, is a sub-command parser from the argparse module. For more details, refer to the documentation for argparse.ArgumentParser.add_subparsers().

The lone common optional argument is:

- –dt-flash (if the runner capabilities includes flash_addr)

Runner-specific options are added through the do_add_parser() hook.

`property build_conf:` runners.core.BuildConfiguration

Get a BuildConfiguration for the build directory.

`call(`*cmd: List[str]*, ***kwargs*`)` → int

Subclass subprocess.call() wrapper.

Subclasses should use this method to run command in a subprocess and get its return code, rather than using subprocess directly, to keep accurate debug logs.

`classmethod capabilities()` → *runners.core.RunnerCaps*

Returns a RunnerCaps representing this runner's capabilities.

This implementation returns the default capabilities.

Subclasses should override appropriately if needed.

`cfg`

RunnerConfig for this instance.

`check_call(`*cmd: List[str]*, ***kwargs*`)`

Subclass subprocess.check_call() wrapper.

Subclasses should use this method to run command in a subprocess and check that it executed correctly, rather than using subprocess directly, to keep accurate debug logs.

`check_output(`*cmd: List[str]*, ***kwargs*`)` → bytes

Subclass subprocess.check_output() wrapper.

Subclasses should use this method to run command in a subprocess and check that it executed correctly, rather than using subprocess directly, to keep accurate debug logs.

`classmethod create(`*cfg:* runners.core.RunnerConfig, *args: argparse.Namespace*`)` → *runners.core.ZephyrBinaryRunner*

Create an instance from command-line arguments.

- `cfg`: runner configuration (pass to superclass __init__)
- `args`: arguments parsed from execution environment, as specified by `add_parser()`.

`abstract classmethod do_add_parser(parser)`

Hook for adding runner-specific options.

`abstract classmethod do_create(cfg: runners.core.RunnerConfig, args: argparse.Namespace) → runners.core.ZephyrBinaryRunner`

Hook for instance creation from command line arguments.

`abstract do_run(command: str, **kwargs)`

Concrete runner; run() delegates to this. Implement in subclasses.

In case of an unsupported command, raise a ValueError.

`ensure_output(output_type: str) → None`

Ensure self.cfg has a particular output artifact.

For example, ensure_output('bin') ensures that self.cfg.bin_file refers to an existing file. Errors out if it's missing or undefined.

Parameters `output_type` – string naming the output type

`static flash_address_from_build_conf(build_conf: runners.core.BuildConfiguration)`

If CONFIG_HAS_FLASH_LOAD_OFFSET is n in build_conf, return the CONFIG_FLASH_BASE_ADDRESS value. Otherwise, return CONFIG_FLASH_BASE_ADDRESS + CONFIG_FLASH_LOAD_OFFSET.

`static get_flash_address(args: argparse.Namespace, build_conf: runners.core.BuildConfiguration, default: int = 0) → int`

Helper method for extracting a flash address.

If args.dt_flash is true, returns the address obtained from ZephyrBinaryRunner.flash_address_from_build_conf(build_conf).

Otherwise (when args.dt_flash is False), the default value is returned.

`static get_runners() → List[Type[runners.core.ZephyrBinaryRunner]]`

Get a list of all currently defined runner classes.

`logger`

logging.Logger for this instance.

`abstract classmethod name() → str`

Return this runner's user-visible name.

When choosing a name, pick something short and lowercase, based on the name of the tool (like openocd, jlink, etc.) or the target architecture/board (like xtensa etc.).

`popen_ignore_int(cmd: List[str]) → subprocess.Popen`

Spawn a child command, ensuring it ignores SIGINT.

The returned subprocess.Popen object must be manually terminated.

`static require(program: str) → str`

Require that a program is installed before proceeding.

Parameters `program` – name of the program that is required, or path to a program binary.

If `program` is an absolute path to an existing program binary, this call succeeds. Otherwise, try to find the program by name on the system PATH.

If the program can be found, its path is returned. Otherwise, raises MissingProgram.

`run(command: str, **kwargs)`

Runs command ('flash', 'debug', 'debugserver', 'attach').

This is the main entry point to this runner.

`run_client`(*client*)

Run a client that handles SIGINT.

`run_server_and_client`(*server*, *client*)

Run a server that ignores SIGINT, and a client that handles it.

This routine portably:

- creates a Popen object for the `server` command which ignores SIGINT
- runs `client` in a subprocess while temporarily ignoring SIGINT
- cleans up the server after the client exits.

It's useful to e.g. open a GDB server and client.

`property thread_info_enabled: bool`

Returns True if self.build_conf has CONFIG_DEBUG_THREAD_INFO enabled. This supports the CONFIG_OPENOCD_SUPPORT fallback as well for now.

Doing it By Hand

If you prefer not to use West to flash or debug your board, simply inspect the build directory for the binaries output by the build system. These will be named something like `zephyr/zephyr.elf`, `zephyr/zephyr.hex`, etc., depending on your board's build system integration. These binaries may be flashed to a board using alternative tools of your choice, or used for debugging as needed, e.g. as a source of symbol tables.

By default, these West commands rebuild binaries before flashing and debugging. This can of course also be accomplished using the usual targets provided by Zephyr's build system (in fact, that's how these commands do it).

8.20.11 Signing Binaries

The `west sign` *extension* command can be used to sign a Zephyr application binary for consumption by a bootloader using an external tool. Run `west sign -h` for command line help.

MCUboot / imgtool

The Zephyr build system has special support for signing binaries for use with the MCUboot bootloader using the imgtool program provided by its developers. You can both build and sign this type of application binary in one step by setting some Kconfig options. If you do, `west flash` will use the signed binaries.

If you use this feature, you don't need to run `west sign` yourself; the build system will do it for you.

Here is an example workflow, which builds and flashes MCUboot, as well as the hello_world application for chain-loading by MCUboot. Run these commands from the `zephyrproject` workspace you created in the *Getting Started Guide*.

```
west build -b YOUR_BOARD -s bootloader/mcuboot/boot/zephyr -d build-mcuboot
west build -b YOUR_BOARD -s zephyr/samples/hello_world -d build-hello-signed -- \
     -DCONFIG_BOOTLOADER_MCUBOOT=y \
     -DCONFIG_MCUBOOT_SIGNATURE_KEY_FILE=\"bootloader/mcuboot/root-rsa-2048.pem\"

west flash -d build-mcuboot
west flash -d build-hello-signed
```

Notes on the above commands:

- `YOUR_BOARD` should be changed to match your board

- The `CONFIG_MCUBOOT_SIGNATURE_KEY_FILE` value is the insecure default provided and used by by MCUboot for development and testing
- You can change the `hello_world` application directory to any other application that can be loaded by MCUboot, such as the smp_svr_sample

For more information on these and other related configuration options, see:

- `CONFIG_BOOTLOADER_MCUBOOT`: build the application for loading by MCUboot
- `CONFIG_MCUBOOT_SIGNATURE_KEY_FILE`: the key file to use with `west sign`. If you have your own key, change this appropriately
- `CONFIG_MCUBOOT_EXTRA_IMGTOOL_ARGS`: optional additional command line arguments for `imgtool`
- `CONFIG_MCUBOOT_GENERATE_CONFIRMED_IMAGE`: also generate a confirmed image, which may be more useful for flashing in production environments than the OTA-able default image
- On Windows, if you get "Access denied" issues, the recommended fix is to run `pip3 install imgtool`, then retry with a pristine build directory.

If your `west flash` *runner* uses an image format supported by imgtool, you should see something like this on your device's serial console when you run `west flash -d build-mcuboot`:

```
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
[00:00:00.004,669] <inf> mcuboot: Starting bootloader
[00:00:00.011,169] <inf> mcuboot: Primary image: magic=unset, swap_type=0x1, copy_
↪done=0x3, image_ok=0x3
[00:00:00.021,636] <inf> mcuboot: Boot source: none
[00:00:00.027,313] <wrn> mcuboot: Failed reading image headers; Image=0
[00:00:00.035,064] <err> mcuboot: Unable to find bootable image
```

Then, you should see something like this when you run `west flash -d build-hello-signed`:

```
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
[00:00:00.004,669] <inf> mcuboot: Starting bootloader
[00:00:00.011,169] <inf> mcuboot: Primary image: magic=unset, swap_type=0x1, copy_
↪done=0x3, image_ok=0x3
[00:00:00.021,636] <inf> mcuboot: Boot source: none
[00:00:00.027,374] <inf> mcuboot: Swap type: none
[00:00:00.115,142] <inf> mcuboot: Bootloader chainload address offset: 0xc000
[00:00:00.123,168] <inf> mcuboot: Jumping to the first image slot
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
Hello World! nrf52840dk_nrf52840
```

Whether `west flash` supports this feature depends on your runner. The `nrfjprog` and `pyocd` runners work with the above flow. If your runner does not support this flow and you would like it to, please send a patch or file an issue for adding support.

8.20.12 Additional Zephyr extension commands

This page documents miscellaneous *Zephyr Extensions*.

Listing boards: `west boards`

The `boards` command can be used to list the boards that are supported by Zephyr without having to resort to additional sources of information.

It can be run by typing:

```
west boards
```

This command lists all supported boards in a default format. If you prefer to specify the display format yourself you can use the `--format` (or `-f`) flag:

```
west boards -f "{arch}:{name}"
```

Additional help about the formatting options can be found by running:

```
west boards -h
```

Installing CMake packages: `west zephyr-export`

This command registers the current Zephyr installation as a CMake config package in the CMake user package registry.

In Windows, the CMake user package registry is found in `HKEY_CURRENT_USER\Software\Kitware\CMake\Packages`.

In Linux and MacOS, the CMake user package registry is found in. `~/.cmake/packages`.

You may run this command when setting up a Zephyr workspace. If you do, application CMakeLists.txt files that are outside of your workspace will be able to find the Zephyr repository with the following:

```
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

See share/zephyr-package/cmake for details.

Software bill of materials: `west spdx`

This command generates SPDX 2.2 tag-value documents, creating relationships from source files to the corresponding generated build files. `SPDX-License-Identifier` comments in source files are scanned and filled into the SPDX documents.

To use this command:

1. Pre-populate a build directory `BUILD_DIR` like this:

   ```
   west spdx --init -d BUILD_DIR
   ```

   This step ensures the build directory contains CMake metadata required for SPDX document generation.

2. Build your application using this pre-created build directory, like so:

   ```
   west build -d BUILD_DIR [...]
   ```

3. Generate SPDX documents using this build directory:

   ```
   west spdx -d BUILD_DIR
   ```

This generates the following SPDX bill-of-materials (BOM) documents in `BUILD_DIR/spdx/`:

- `app.spdx`: BOM for the application source files used for the build
- `zephyr.spdx`: **BOM for the specific Zephyr source code files used for** the build
- `build.spdx`: BOM for the built output files

Each file in the bill-of-materials is scanned, so that its hashes (SHA256 and SHA1) can be recorded, along with any detected licenses if an `SPDX-License-Identifier` comment appears in the file.

SPDX Relationships are created to indicate dependencies between CMake build targets, build targets that are linked together, and source files that are compiled to generate the built library files.

`west spdx` accepts these additional options:

- `-n PREFIX`: a prefix for the Document Namespaces that will be included in the generated SPDX documents. See SPDX specification 2.2 section 2.5 for details. If `-n` is omitted, a default namespace will be generated according to the default format described in section 2.5 using a random UUID.
- `-s SPDX_DIR`: specifies an alternate directory where the SPDX documents should be written instead of `BUILD_DIR/spdx/`.
- `--analyze-includes`: in addition to recording the compiled source code files (e.g. `.c`, `.S`) in the bills-of-materials, also attempt to determine the specific header files that are included for each `.c` file.

  This takes longer, as it performs a dry run using the C compiler for each `.c` file using the same arguments that were passed to it for the actual build.
- `--include-sdk`: with `--analyze-includes`, also create a fourth SPDX document, `sdk.spdx`, which lists header files included from the SDK.

8.20.13 History and Motivation

West was added to the Zephyr project to fulfill two fundamental requirements:

- The ability to work with multiple Git repositories
- The ability to provide an extensible and user-friendly command-line interface for basic Zephyr workflows

During the development of west, a set of *Design Constraints* were identified to avoid the common pitfalls of tools of this kind.

Requirements

Although the motivation behind splitting the Zephyr codebase into multiple repositories is outside of the scope of this page, the fundamental requirements, along with a clear justification of the choice not to use existing tools and instead develop a new one, do belong here.

The basic requirements are:

- **R1**: Keep externally maintained code in separately maintained repositories outside of the main zephyr repository, without requiring users to manually clone each of the external repositories
- **R2**: Provide a tool that both Zephyr users and distributors can make use of to benefit from and extend
- **R3**: Allow users and downstream distributions to override or remove repositories without having to make changes to the zephyr repository
- **R4**: Support both continuous tracking and commit-based (bisectable) project updating

Rationale for a custom tool

Some of west's features are similar to those provided by Git Submodules and Google's repo.

Existing tools were considered during west's initial design and development. None were found suitable for Zephyr's requirements. In particular, these were examined in detail:

- Google repo

  - Does not cleanly support using zephyr as the manifest repository (**R4**)
  - Python 2 only
  - Does not play well with Windows
  - Assumes Gerrit is used for code review
- Git submodules
  - Does not fully support **R1**, since the externally maintained repositories would still need to be inside the main zephyr Git tree
  - Does not support **R3**, since downstream copies would need to either delete or replace submodule definitions
  - Does not support continuous tracking of the latest `HEAD` in external repositories (**R4**)
  - Requires hardcoding of the paths/locations of the external repositories

Multiple Git Repositories

Zephyr intends to provide all required building blocks needed to deploy complex IoT applications. This in turn means that the Zephyr project is much more than an RTOS kernel, and is instead a collection of components that work together. In this context, there are a few reasons to work with multiple Git repositories in a standardized manner within the project:

- Clean separation of Zephyr original code and imported projects and libraries
- Avoidance of license incompatibilities between original and imported code
- Reduction in size and scope of the core Zephyr codebase, with additional repositories containing optional components instead of being imported directly into the tree
- Safety and security certifications
- Enforcement of modularization of the components
- Out-of-tree development based on subsets of the supported boards and SoCs

See *Basics* for information on how west workspaces manage multiple git repositories.

Design Constraints

West is:

- **Optional**: it is always *possible* to drop back to "raw" command-line tools, i.e. use Zephyr without using west (although west itself might need to be installed and accessible to the build system). It may not always be *convenient* to do so, however. (If all of west's features were already conveniently available, there would be no reason to develop it.)
- **Compatible with CMake**: building, flashing and debugging, and emulator support will always remain compatible with direct use of CMake.
- **Cross-platform**: West is written in Python 3, and works on all platforms supported by Zephyr.
- **Usable as a Library**: whenever possible, west features are implemented as libraries that can be used standalone in other programs, along with separate command line interfaces that wrap them. West itself is a Python package named `west`; its libraries are implemented as subpackages.
- **Conservative about features**: no features will be accepted without strong and compelling motivation.
- **Clearly specified**: West's behavior in cases where it wraps other commands is clearly specified and documented. This enables interoperability with third party tools, and means Zephyr developers can always find out what is happening "under the hood" when using west.

See Zephyr issue #6205 and for more details and discussion.

8.20.14 Moving to West

To convert a "pre-west" Zephyr setup on your computer to west, follow these steps. If you are starting from scratch, use the *Getting Started Guide* instead. See *Troubleshooting West* for advice on common issues.

1. Install west.

   On Linux:

   ```
   pip3 install --user -U west
   ```

   On Windows and macOS:

   ```
   pip3 install -U west
   ```

   For details, see *Installing west*.

2. Move your zephyr repository to a new `zephyrproject` parent directory, and change directory there.

   On Linux and macOS:

   ```
   mkdir zephyrproject
   mv zephyr zephyrproject
   cd zephyrproject
   ```

   On Windows `cmd.exe`:

   ```
   mkdir zephyrproject
   move zephyr zephyrproject
   chdir zephyrproject
   ```

   The name `zephyrproject` is recommended, but you can choose any name with no spaces anywhere in the path.

3. Create a *west workspace* using the zephyr repository as a local manifest repository:

   ```
   west init -l zephyr
   ```

   This creates `zephyrproject/.west`, marking the root of your workspace, and does some other setup. It will not change the contents of the zephyr repository in any way.

4. Clone the rest of the repositories used by zephyr:

   ```
   west update
   ```

   Make sure to run this command whenever you pull zephyr. Otherwise, your local repositories will get out of sync. (Run `west list` for current information on these repositories.)

You are done: `zephyrproject` is now set up to use west.

8.20.15 Using Zephyr without west

This page provides information on using Zephyr without west. This is not recommended for beginners due to the extra effort involved. In particular, you will have to do work "by hand" to replace these features:

- cloning the additional source code repositories used by Zephyr in addition to the main zephyr repository, and keeping them up to date

- specifying the locations of these repositories to the Zephyr build system
- flashing and debugging without understanding detailed usage of the relevant host tools

---

Note: If you have previously installed west and want to stop using it, uninstall it first:

```
pip3 uninstall west
```

Otherwise, Zephyr's build system will find it and may try to use it.

Getting the Source

In addition to downloading the zephyr source code repository itself, you will need to manually clone the additional projects listed in the west manifest file inside that repository.

```
mkdir zephyrproject
cd zephyrproject
git clone https://github.com/zephyrproject-rtos/zephyr
# clone additional repositories listed in zephyr/west.yml,
# and check out the specified revisions as well.
```

As you pull changes in the zephyr repository, you will also need to maintain those additional repositories, adding new ones as necessary and keeping existing ones up to date at the latest revisions.

Building applications

You can build a Zephyr application using CMake and Ninja (or make) directly without west installed if you specify any modules manually.

```
cmake -B build -GNinja -DZEPHYR_MODULES=module1;module2;... samples/hello_world
ninja -C build
```

When building with west installed, the Zephyr build system will use it to set *ZEPHYR_MODULES*.

If you don't have west installed and your application does not need any of these repositories, the build will still work.

If you don't have west installed and your application *does* need one of these repositories, you must set `ZEPHYR_MODULES` yourself as shown above.

See *Modules (External projects)* for more details.

Flashing and Debugging

Running build system targets like `ninja flash`, `ninja debug`, etc. is just a call to the corresponding *west command*. For example, `ninja flash` calls `west flash`[1]. If you don't have west installed on your system, running those targets will fail. You can of course still flash and debug using any *Flash & Debug Host Tools* which work for your board (and which those west commands wrap).

If you want to use these build system targets but do not want to install west on your system using `pip`, it is possible to do so by manually creating a west workspace:

[1] Note that `west build` invokes `ninja`, among other tools. There's no recursive invocation of either `west` or `ninja` involved by default, however, as `west build` does not invoke `ninja flash`, `debug`, etc. The one exception is if you specifically run one of these build system targets with a command line like `west build -t flash`. In that case, west is run twice: once for `west build`, and in a subprocess, again for `west flash`. Even in this case, `ninja` is only run once, as `ninja flash`. This is because these build system targets depend on an up to date build of the Zephyr application, so it's compiled before `west flash` is run.

```
# cd into zephyrproject if not already there
git clone https://github.com/zephyrproject-rtos/west.git .west/west
```

Then create a file `.west/config` with the following contents:

```
[manifest]
path = zephyr

[zephyr]
base = zephyr
```

After that, and in order for `ninja` to be able to invoke `west` to flash and debug, you must specify the west directory. This can be done by setting the environment variable `WEST_DIR` to point to `zephyrproject/.west/west` before running CMake to set up a build directory.

For details on west's Python APIs, see west-apis.

8.21 Optimizations

Guides on how to optimize Zephyr for performance, power and footprint.

8.21.1 Optimizing for Footprint

Stack Sizes

Stack sizes of various system threads are specified generously to allow for usage in different scenarios on as many supported platforms as possible. You should start the optimization process by reviewing all stack sizes and adjusting them for your application:

`CONFIG_ISR_STACK_SIZE` Set to 2048 by default

`CONFIG_MAIN_STACK_SIZE` Set to 1024 by default

`CONFIG_IDLE_STACK_SIZE` Set to 320 by default

`CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE` Set to 1024 by default

`CONFIG_PRIVILEGED_STACK_SIZE` Set to 1024 by default, depends on userspace feature.

Unused Peripherals

Some peripherals are enabled by default. You can disable unused peripherals in your project configuration, for example:

```
CONFIG_GPIO=n
CONFIG_SPI=n
```

Various Debug/Informational Options

The following options are enabled by default to provide more information about the running application and to provide means for debugging and error handling:

`CONFIG_BOOT_BANNER` This option can be disabled to save a few bytes.

`CONFIG_DEBUG` This option can be disabled for production builds

MPU/MMU Support

Depending on your application and platform needs, you can disable MPU/MMU support to gain some memory and improve performance. Consider the consequences of this configuration choice though, because you'll lose advanced stack checking and support.

8.21.2 Optimization Tools

Footprint and Memory Usage

The build system offers 3 targets to view and analyse RAM, ROM and stack usage in generated images. The tools run on the final image and give information about size of symbols and code being used in both RAM and ROM. Additionally, with features available through the compiler, we can also generate worst-case stack usage analysis:

Tools that are available as build system targets:

Build Target: puncover This target uses a 3rd party tools called puncover which can be found here. When this target is built, it will launch a local web server which will allow you to open a web client and browse the files and view their ROM, RAM and stack usage. Before you can use this target, you will have to install the puncover python module:

```
pip3 install git+https://github.com/HBehrens/puncover --user
```

Then:

Using west:

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

Using CMake and ninja:

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

# Now run ninja on the generated build system:
ninja -C build puncover
```

To view worst-case stack usage analysis, build this with the `CONFIG_STACK_USAGE` enabled.

Using west:

```
west build -b reel_board samples/hello_world -- -DCONFIG_STACK_USAGE=y
west build -t puncover
```

Using CMake and ninja:

```
# Use cmake to configure a Ninja-based buildsystem:
cmake -B build -GNinja -DBOARD=reel_board -DCONFIG_STACK_USAGE=y samples/hello_world

# Now run ninja on the generated build system:
ninja -C build puncover
```

Build Target: ram_report List all compiled objects and their RAM usage in a tabular form with bytes per symbol and the percentage it uses. The data is grouped based on the file system location of the object in the tree and the file containing the symbol.

Use the `ram_report` target with your board:

Using west:

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

Using CMake and ninja:

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

# Now run ninja on the generated build system:
ninja -C build ram_report
```

which will generate something similar to the output below:

```
Path                                                                                                                ␣
↪  Size       %
====================================================================================================================
...
...
SystemCoreClock                                                                                                     ␣
↪    4     0.08%
_kernel                                                                                                             ␣
↪   48     0.99%
_sw_isr_table                                                                                                       ␣
↪  384     7.94%
cli.10544                                                                                                           ␣
↪   16     0.33%
gpio_initialized.9765                                                                                               ␣
↪    1     0.02%
on.10543                                                                                                            ␣
↪    4     0.08%
poll_out_lock.9764                                                                                                  ␣
↪    4     0.08%
z_idle_threads                                                                                                      ␣
↪  128     2.65%
z_interrupt_stacks                                                                                                  ␣
↪2048    42.36%
z_main_thread                                                                                                       ␣
↪  128     2.65%
arch                                                                                                                ␣
↪      1     0.02%
arm                                                                                                                 ␣
↪    1     0.02%
    core                                                                                                            ␣
↪      1     0.02%
    aarch32                                                                                                         ␣
↪    1     0.02%
        cortex_m                                                                                                    ␣
↪      1     0.02%
        mpu                                                                                                         ␣
↪    1     0.02%
            arm_mpu.c                                                                                               ␣
↪      1     0.02%
            static_regions_num                                                                                      ␣
↪    1     0.02%
drivers                                                                                                             ␣
↪    536    11.09%
```

(continues on next page)

(continued from previous page)

```
clock_control                                                                 ␣
↪ 100     2.07%
    nrf_power_clock.c                                                         ␣
↪   100     2.07%
    __device_clock_nrf                                                        ␣
↪  16     0.33%
    data                                                                      ␣
↪  80     1.65%
    hfclk_users                                                               ␣
↪   4     0.08%
...
...
```

Build Target: rom_report List all compiled objects and their ROM usage in a tabular form with bytes per symbol and the percentage it uses. The data is grouped based on the file system location of the object in the tree and the file containing the symbol.

Use the `rom_report` to get the ROM report:

Using west:

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

Using CMake and ninja:

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

# Now run ninja on the generated build system:
ninja -C build rom_report
```

which will generate something similar to the output below:

```
Path                                                                          ␣
↪  Size       %
==============================================================================================================
...
...
CSWTCH.5                                                                      ␣
↪   4     0.02%
SystemCoreClock                                                               ␣
↪   4     0.02%
__aeabi_idiv0                                                                 ␣
↪   2     0.01%
__udivmoddi4                                                                  ␣
↪ 702     3.37%
_sw_isr_table                                                                 ␣
↪ 384     1.85%
delay_machine_code.9114                                                       ␣
↪   6     0.03%
levels.8826                                                                   ␣
↪  20     0.10%
mpu_config                                                                    ␣
↪   8     0.04%
transitions.10558                                                             ␣
↪  12     0.06%
```

(continues on next page)

(continued from previous page)

```
arch                                                                            ␣
↪  1194     5.74%
arm                                                                             ␣
↪1194     5.74%
    core                                                                        ␣
↪  1194     5.74%
    aarch32                                                                     ␣
↪1194     5.74%
        cortex_m                                                                ␣
↪   852     4.09%
        fault.c                                                                 ␣
↪ 400     1.92%
            bus_fault.isra.0                                                    ␣
↪    60     0.29%
            mem_manage_fault.isra.0                                             ␣
↪    56     0.27%
            usage_fault.isra.0                                                  ␣
↪    36     0.17%
            z_arm_fault                                                         ␣
↪   232     1.11%
            z_arm_fault_init                                                    ␣
↪    16     0.08%
        irq_init.c                                                              ␣
↪  24     0.12%
            z_arm_interrupt_init                                                ␣
↪    24     0.12%
        mpu                                                                     ␣
↪ 352     1.69%
            arm_core_mpu.c                                                      ␣
↪    56     0.27%
            z_arm_configure_static_mpu_regions                                  ␣
↪  56     0.27%
            arm_mpu.c                                                           ␣
↪   296     1.42%
            __init_sys_init_arm_mpu_init0                                       ␣
↪   8     0.04%
            arm_core_mpu_configure_static_mpu_regions                           ␣
↪  20     0.10%
            arm_core_mpu_disable                                                ␣
↪  16     0.08%
            arm_core_mpu_enable                                                 ␣
↪  20     0.10%
            arm_mpu_init                                                        ␣
↪  92     0.44%
            mpu_configure_regions                                               ␣
↪ 140     0.67%
        thread_abort.c                                                          ␣
↪  76     0.37%
            z_impl_k_thread_abort
            76     0.37%
...
...
```

Data Structures

Build Target: pahole Poke-a-hole (pahole) is an object-file analysis tool to find the size of the data structures, and the holes caused due to aligning the data elements to the word-size of the CPU by the compiler.

Poke-a-hole (pahole) must be installed prior to using this target. It can be obtained from https://git.kernel.org/pub/scm/devel/pahole/pahole.git and is available in the dwarves package in both fedora and ubuntu:

```
sudo apt-get install dwarves
```

or in fedora:

```
sudo dnf install dwarves
```

Using west:

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

Using CMake and ninja:

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

# Now run ninja on the generated build system:
ninja -C build pahole
```

After running this target, pahole will output the results to the console:

```
/* Used at: zephyr/isr_tables.c */
/* <80> ../include/sw_isr_table.h:30 */
struct _isr_table_entry {
        void *                     arg;                  /*     0     4 */
        void                       (*isr)(void *);       /*     4     4 */

        /* size: 8, cachelines: 1, members: 2 */
        /* last cacheline: 8 bytes */
};
/* Used at: zephyr/isr_tables.c */
/* <eb> ../include/arch/arm/aarch32/cortex_m/mpu/arm_mpu_v7m.h:134 */
struct arm_mpu_region_attr {
        uint32_t                   rasr;                 /*     0     4 */

        /* size: 4, cachelines: 1, members: 1 */
        /* last cacheline: 4 bytes */
};
/* Used at: zephyr/isr_tables.c */
/* <112> ../include/arch/arm/aarch32/cortex_m/mpu/arm_mpu.h:24 */
struct arm_mpu_region {
        uint32_t                   base;                 /*     0     4 */
        const char  *              name;                 /*     4     4 */
        arm_mpu_region_attr_t      attr;                 /*     8     4 */

        /* size: 12, cachelines: 1, members: 3 */
        /* last cacheline: 12 bytes */
};
...
...
```

8.22 Zephyr CMake Package

The Zephyr CMake package is a convenient way to create a Zephyr-based application.

The Zephyr CMake package ensures that CMake can automatically select a Zephyr to use for building the application, whether it is a Zephyr repository application, Zephyr workspace application, or a Zephyr freestanding application.

When developing a Zephyr-based application, then a developer simply needs to write `find_package(Zephyr)` in the beginning of the application `CMakeLists.txt` file.

To use the Zephyr CMake package it must first be exported to the CMake user package registry. This is means creating a reference to the current Zephyr installation inside the CMake user package registry.

Ubuntu

In Linux, the CMake user package registry is found in:

`~/.cmake/package/Zephyr`

macOS

In macOS, the CMake user package registry is found in:

`~/.cmake/package/Zephyr`

Windows

In Windows, the CMake user package registry is found in:

`HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\Zephyr`

The Zephyr CMake package allows CMake to automatically find a Zephyr base. One or more Zephyr installations must be exported. Exporting multiple Zephyr installations may be useful when developing or testing Zephyr freestanding applications, Zephyr workspace application with vendor forks, etc..

8.22.1 Zephyr CMake package export (west)

When installing Zephyr using *west* then it is recommended to export Zephyr using `west zephyr-export`.

8.22.2 Zephyr CMake package export (without west)

Zephyr CMake package is exported to the CMake user package registry using the following commands:

```
cmake -P <PATH-TO-ZEPHYR>/share/zephyr-package/cmake/zephyr_export.cmake
```

This will export the current Zephyr to the CMake user package registry.

To also export the Zephyr Unittest CMake package, run the following command in addition:

```
cmake -P <PATH-TO-ZEPHYR>/share/zephyrunittest-package/cmake/zephyr_export.cmake
```

8.22.3 Zephyr application structure

An application can be placed anywhere on your disk, but to better understand how the Zephyr package is used, we will name three specific layouts.

Zephyr repository application

A Zephyr repository has the following structure:

```
<projects>/zephyr-workspace
└── zephyr
    ├── arch
    ├── boards
    ├── cmake
    ├── samples
    │   ├── hello_world
    │   └── ...
    ├── tests
    └── ...
```

Any application located inside this tree, is simply referred to as a Zephyr repository application. In this example `hello_world` is a Zephyr repository application.

Zephyr workspace application

A Zephyr workspace has the following structure:

```
<projects>/zephyr-workspace
├── zephyr
├── bootloader
├── modules
├── tools
├── <vendor/private-repositories>
└── my_applications
    └── my_first_app
```

Any application located in such workspace, but outside the Zephyr repository itself, is referred to as a Zephyr workspace application. In this example `my_first_app` is a Zephyr workspace application.

Note: The root of a Zephyr workspace is identical to `west topdir` if the workspace was installed using `west`

Zephyr freestanding application

A Zephyr freestanding application is a Zephyr application located outside of a Zephyr workspace.

```
<projects>/zephyr-workspace
├── zephyr
├── bootloader
└── ...

<home>/app
├── CMakeLists.txt
├── prj.conf
└── src
    └── main.c
```

In this example `app` is a Zephyr freestanding application.

8.22.4 Zephyr Base Environment Setting

The Zephyr CMake package search functionality allows for explicitly specifying a Zephyr base using an environment variable.

To do this, use the following `find_package()` syntax:

```
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

This syntax instructs CMake to first search for Zephyr using the Zephyr base environment setting `ZEPHYR_BASE` and then use the normal search paths.

8.22.5 Zephyr CMake Package Search Order

When Zephyr base environment setting is not used for searching, the Zephyr installation matching the following criteria will be used:

- A Zephyr repository application will use the Zephyr in which it is located. For example:

  ```
  <projects>/zephyr-workspace/zephyr
  └── samples
      └── hello_world
  ```

  in this example, `hello_world` will use `<projects>/zephyr-workspace/zephyr`.

- Zephyr workspace application will use the Zephyr that share the same workspace. For example:

  ```
  <projects>/zephyr-workspace
  ├── zephyr
  ├── ...
  └── my_applications
       └── my_first_app
  ```

  in this example, `my_first_app` will use `<projects>/zephyr-workspace/zephyr` as this Zephyr is located in the same workspace as the Zephyr workspace application.

- Zephyr freestanding application will use the Zephyr registered in the CMake user package registry. For example:

  ```
  <projects>/zephyr-workspace-1
  └── zephyr                          (Not exported to CMake)

  <projects>/zephyr-workspace-2
  └── zephyr                          (Exported to CMake)

  <home>/app
  ├── CMakeLists.txt
  ├── prj.conf
  └── src
      └── main.c
  ```

  in this example, only `<projects>/zephyr-workspace-2/zephyr` is exported to the CMake package registry and therefore this Zephyr will be used by the Zephyr freestanding application `<home>/app`.

  If user wants to test the application with `<projects>/zephyr-workspace-1/zephyr`, this can be done by using the Zephyr Base environment setting, meaning set `ZEPHYR_BASE=<projects>/zephyr-workspace-1/zephyr`, before running CMake.

Note: The Zephyr package selected on the first CMake invocation will be used for all subsequent builds. To change the Zephyr package, for example to test the application using Zephyr base

environment setting, then it is necessary to do a pristine build first (See *Rebuilding an Application*).

8.22.6 Zephyr CMake Package Version

When writing an application then it is possible to specify a Zephyr version number `x.y.z` that must be used in order to build the application.

Specifying a version is especially useful for a Zephyr freestanding application as it ensures the application is built with a minimal Zephyr version.

It also helps CMake to select the correct Zephyr to use for building, when there are multiple Zephyr installations in the system.

For example:

```
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr 2.2.0)
project(app)
```

will require app to be built with Zephyr 2.2.0 as minimum. CMake will search all exported candidates to find a Zephyr installation which matches this version criteria.

Thus it is possible to have multiple Zephyr installations and have CMake automatically select between them based on the version number provided, see CMake package version for details.

For example:

```
<projects>/zephyr-workspace-2.a
└── zephyr                        (Exported to CMake)

<projects>/zephyr-workspace-2.b
└── zephyr                        (Exported to CMake)

<home>/app
├── CMakeLists.txt
├── prj.conf
└── src
    └── main.c
```

in this case, there are two released versions of Zephyr installed at their own workspaces. Workspace 2.a and 2.b, corresponding to the Zephyr version.

To ensure app is built with minimum version `2.a` the following `find_package` syntax may be used:

```
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr 2.a)
project(app)
```

Note that both `2.a` and `2.b` fulfill this requirement.

CMake also supports the keyword `EXACT`, to ensure an exact version is used, if that is required. In this case, the application CMakeLists.txt could be written as:

```
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr 2.a EXACT)
project(app)
```

In case no Zephyr is found which satisfies the version required, as example, the application specifies

```
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr 2.z)
project(app)
```

then an error similar to below will be printed:

```
Could not find a configuration file for package "Zephyr" that is compatible
with requested version "2.z".

The following configuration files were considered but not accepted:

  <projects>/zephyr-workspace-2.a/zephyr/share/zephyr-package/cmake/ZephyrConfig.
↪cmake, version: 2.a.0
  <projects>/zephyr-workspace-2.b/zephyr/share/zephyr-package/cmake/ZephyrConfig.
↪cmake, version: 2.b.0
```

Note: It can also be beneficial to specify a version number for Zephyr repository applications and Zephyr workspace applications. Specifying a version in those cases ensures the application will only build if the Zephyr repository or workspace is matching. This can be useful to avoid accidental builds when only part of a workspace has been updated.

8.22.7 Multiple Zephyr Installations (Zephyr workspace)

Testing out a new Zephyr version, while at the same time keeping the existing Zephyr in the workspace untouched is sometimes beneficial.

Or having both an upstream Zephyr, Vendor specific, and a custom Zephyr in same workspace.

For example:

```
<projects>/zephyr-workspace
├── zephyr
├── zephyr-vendor
├── zephyr-custom
├── ...
└── my_applications
    └── my_first_app
```

in this setup, `find_package(Zephyr)` has the following order of precedence for selecting which Zephyr to use:

- Project name: `zephyr`
- First project, when Zephyr projects are ordered lexicographical, in this case.
  - `zephyr-custom`
  - `zephyr-vendor`

This means that `my_first_app` will use `<projects>/zephyr-workspace/zephyr`.

It is possible to specify a Zephyr preference list in the application.

A Zephyr preference list can be specified as:

```
cmake_minimum_required(VERSION 3.13.1)

set(ZEPHYR_PREFER "zephyr-custom" "zephyr-vendor")
find_package(Zephyr)
```

(continues on next page)

(continued from previous page)

```
project(my_first_app)
```

the ZEPHYR_PREFER is a list, allowing for multiple Zephyrs. If a Zephyr is specified in the list, but not found in the system, it is simply ignored and `find_package(Zephyr)` will continue to the next candidate.

This allows for temporary creation of a new Zephyr release to be tested, without touching current Zephyr. When testing is done, the `zephyr-test` folder can simply be removed. Such a CMakeLists.txt could look as:

```
cmake_minimum_required(VERSION 3.13.1)

set(ZEPHYR_PREFER "zephyr-test")
find_package(Zephyr)

project(my_first_app)
```

8.22.8 Zephyr Build Configuration CMake package

The Zephyr Build Configuration CMake package provides a possibility for a Zephyr based project to control Zephyr build settings in a generic way.

It is similar to the use of `.zephyrrc` but with the possibility to automatically allow all users to share the build configuration through the project repository. But it also allows more advanced use cases than a `.zephyrrc`-file, such as loading of additional CMake boilerplate code.

The Zephyr Build Configuration CMake package will be loaded in the Zephyr boilerplate code after initial properties and ZEPHYR_BASE has been defined, but before CMake code execution. This allows the Zephyr Build Configuration CMake package to setup or extend properties such as: DTS_ROOT, BOARD_ROOT, TOOLCHAIN_ROOT / other toolchain setup, fixed overlays, and any other property that can be controlled. It also allows inclusion of additional boilerplate code.

To provide a Zephyr Build Configuration CMake package, create `ZephyrBuildConfig.cmake` and place it in a Zephyr workspace top-level folder as:

```
<projects>/zephyr-workspace
├── zephyr
├── ...
└── zephyr application (can be named anything)
    └── share/zephyrbuild-package/cmake/ZephyrBuildConfig.cmake
```

The Zephyr Build Configuration CMake package will not search in any CMake default search paths, and thus cannot be installed in the CMake package registry. There will be no version checking on the Zephyr Build Configuration package.

Note: `share/zephyrbuild-package/cmake/ZephyrBuildConfig.cmake` follows the same folder structure as the Zephyr CMake package.

It is possible to place `ZephyrBuildConfig.cmake` directly in a `<zephyr application>/cmake` folder or another folder, as long as that folder is honoring the CMake package search algorithm.

A sample `ZephyrBuildConfig.cmake` can be seen below.

```
# ZephyrBuildConfig.cmake sample code

# To ensure final path is absolute and does not contain ../.. in variable.
get_filename_component(APPLICATION_PROJECT_DIR
```

(continues on next page)

(continued from previous page)

```
                        ${CMAKE_CURRENT_LIST_DIR}/../../..
                        ABSOLUTE
)

# Add this project to list of board roots
list(APPEND BOARD_ROOT ${APPLICATION_PROJECT_DIR})

# Default to GNU Arm Embedded toolchain if no toolchain is set
if(NOT ENV{ZEPHYR_TOOLCHAIN_VARIANT})
    set(ZEPHYR_TOOLCHAIN_VARIANT gnuarmemb)
    find_program(GNU_ARM_GCC arm-none-eabi-gcc)
    if(NOT ${GNU_ARM_GCC} STREQUAL GNU_ARM_GCC-NOTFOUND)
        # The toolchain root is located above the path to the compiler.
        get_filename_component(GNUARMEMB_TOOLCHAIN_PATH ${GNU_ARM_GCC}/../.. ABSOLUTE)
    endif()
endif()
```

8.22.9 Zephyr Build Configuration CMake package (Freestanding application)

The Zephyr Build Configuration CMake package can be located outside a Zephyr workspace, for example located with a *Zephyr freestanding application*.

Create the build configuration as described in the previous section, and then refer to the location of your Zephyr Build Configuration CMake package using the CMake variable `ZephyrBuildConfiguration_ROOT`.

1. At the CMake command line, like this:

   ```
   cmake -DZephyrBuildConfiguration_ROOT=<path-to-build-config> ...
   ```

2. At the top of your application's top level `CMakeLists.txt`, like this:

   ```
   set(ZephyrBuildConfiguration_ROOT <path-to-build-config>)
   find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
   ```

   If you choose this option, make sure to set the variable **before** calling `find_package(Zephyr ...)`, as shown above.

3. In a separate CMake script which is pre-loaded to populate the CMake cache, like this:

   ```
   # Put this in a file with a name like "zephyr-settings.cmake"
   set(ZephyrBuildConfiguration_ROOT <path-to-build-config>
       CACHE STRING "pre-cached build config"
   )
   ```

   You can tell the build system to use this file by adding `-C zephyr-settings.cmake` to your CMake command line. This principle is useful when not using `west` as both this setting and Zephyr modules can be specified using the same file. See Zephyr module *Without West*.

8.22.10 Zephyr CMake package source code

The Zephyr CMake package source code in `<PATH-TO-ZEPHYR>/share/zephyr-package/cmake` contains the CMake config package which is used by CMake `find_package` function.

It also contains code for exporting Zephyr as a CMake config package.

The following is an overview of those files

`CMakeLists.txt` The CMakeLists.txt file for the CMake build system which is responsible for exporting Zephyr as a package to the CMake user package registry.

`ZephyrConfigVersion.cmake` The Zephyr package version file. This file is called by CMake to determine if this installation fulfils the requirements specified by user when calling `find_package(Zephyr . ..)`. It is also responsible for detection of Zephyr repository or workspace only installations.

`ZephyrUnittestConfigVersion.cmake` Same responsibility as `ZephyrConfigVersion.cmake`, but for unit tests. Includes `ZephyrConfigVersion.cmake`.

`ZephyrConfig.cmake` The Zephyr package file. This file is called by CMake to for the package meeting which fulfils the requirements specified by user when calling `find_package(Zephyr ...)`. This file is responsible for sourcing of boilerplate code.

`ZephyrUnittestConfig.cmake` Same responsibility as `ZephyrConfig.cmake`, but for unit tests. Includes `ZephyrConfig.cmake`.

`zephyr_package_search.cmake` Common file used for detection of Zephyr repository and workspace candidates. Used by `ZephyrConfigVersion.cmake` and `ZephyrConfig.cmake` for common code.

`pristine.cmake` Pristine file for removing all files created by CMake during configure and generator time when exporting Zephyr CMake package. Running pristine keeps all package related files mentioned above.

Chapter 9

Security

These documents describe the requirements, processes, and developer guidelines for ensuring security is addressed within the Zephyr project.

9.1 Zephyr Security Overview

9.1.1 Introduction

This document outlines the steps of the Zephyr Security Subcommittee towards a defined security process that helps developers build more secure software while addressing security compliance requirements. It presents the key ideas of the security process and outlines which documents need to be created. After the process is implemented and all supporting documents are created, this document is a top-level overview and entry point.

Overview and Scope

We begin with an overview of the Zephyr development process, which mainly focuses on security functionality.

In subsequent sections, the individual parts of the process are treated in detail. As depicted in Figure 1, these main steps are:

1. **Secure Development:** Defines the system architecture and development process that ensures adherence to relevant coding principles and quality assurance procedures.
2. **Secure Design:** Defines security procedures and implement measures to enforce them. A security architecture of the system and relevant sub-modules is created, threats are identified, and countermeasures designed. Their correct implementation and the validity of the threat models are checked by code reviews. Finally, a process shall be defined for reporting, classifying, and mitigating security issues..
3. **Security Certification:** Defines the certifiable part of the Zephyr RTOS. This includes an evaluation target, its assets, and how these assets are protected. Certification claims shall be determined and backed with appropriate evidence.

Intended Audience

This document is a guideline for the development of a security process by the Zephyr Security Subcommittee and the Zephyr Technical Steering Committee. It provides an overview of the Zephyr security process for (security) engineers and architects.

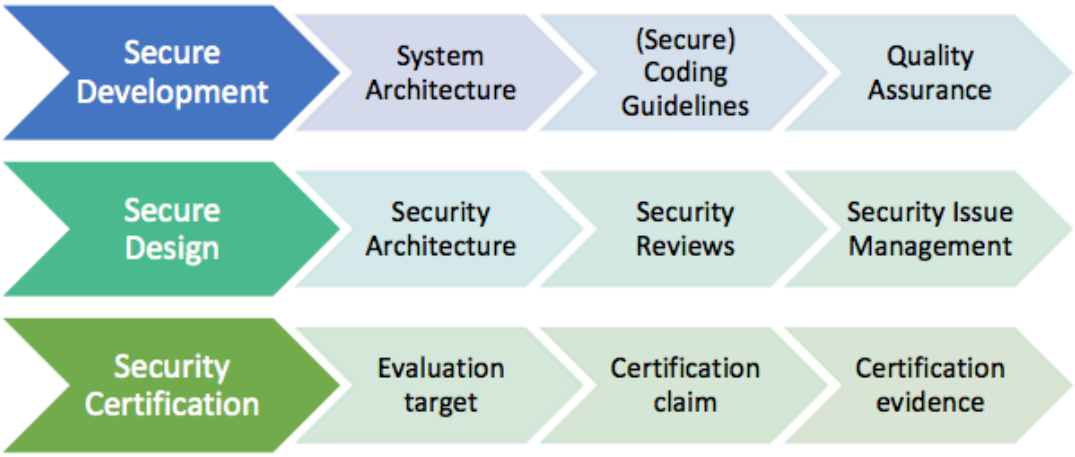


Fig. 1: Figure 1. Security Process Steps

Nomenclature

In this document, the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in [**?**].

These words are used to define absolute requirements (or prohibitions), highly recommended requirements, and truly optional requirements. As noted in RFC-2119, "These terms are frequently used to specify behavior with security implications. The effects on security of not implementing a MUST or SHOULD, or doing something the specification says MUST NOT or SHOULD NOT be done may be very subtle. Document authors should take the time to elaborate the security implications of not following recommendations or requirements as most implementors will not have had the benefit of the experience and discussion that produced the specification."

Security Document Update

This document is a living document. As new requirements, features, and changes are identified, they will be added to this document through the following process:

1. Changes will be submitted from the interested party(ies) via pull requests to the Zephyr documentation repository.
2. The Zephyr Security Subcommittee will review these changes and provide feedback or acceptance of the changes.
3. Once accepted, these changes will become part of the document.

9.1.2 Current Security Definition

This section recapitulates the current status of secure development within the Zephyr RTOS. Currently, focus is put on functional security and code quality assurance, although additional security features are scoped.

The three major security measures currently implemented are:

- **Security Functionality** with a focus on cryptographic algorithms and protocols. Support for cryptographic hardware is scoped for future releases.The Zephyr runtime architecture is a monolithic binary and removes the need for dynamic loaders , thereby reducing the exposed attack surface.

- **Quality Assurance** is driven by using a development process that requires all code to be reviewed before being committed to the common repository. Furthermore, the reuse of proven building blocks such as network stacks increases the overall quality level and guarantees stable APIs. Static code analyses are provided by Coverity Scan.
- **Execution Protection** including thread separation, stack and memory protection is currently available in the upstream Zephyr RTOS starting with version 1.9.0 (stack protection). Memory protection and thread separation was added in version 1.10.0 for X86 and in version 1.11.0 for ARM and ARC.

These topics are discussed in more detail in the following subsections.

Security Functionality

The security functionality in Zephyr hinges mainly on the inclusion of cryptographic algorithms, and on its monolithic system design.

The cryptographic features are provided through a set of cryptographic libraries. Applications can choose TinyCrypt2 or mbedTLS based on their needs. TinyCrypt2 supports key cryptographic algorithms required by the connectivity stacks. Tinycrypt2, however, only provides a limited set of algorithms. mbedTLS supports a wider range of algorithms, but at the cost of additional requirements such as malloc support. Applications can choose the solution that matches their individual requirements. Future work may include APIs to abstract the underlying crypto library choice.

APIs for vendor specific cryptographic IPs in both hardware and software are planned, including secure key storage in the form of secure access modules (SAMs), Trusted Platform Modules (TPMs), and Trusted Execution Environments (TEEs).

The security architecture is based on a monolithic design where the Zephyr kernel and all applications are compiled into a single static binary. System calls are implemented as function calls without requiring context switches. Static linking eliminates the potential for dynamically loading malicious code.

Additional protection features are available in later releases. Stack protection mechanisms are provided to protect against stack overruns. In addition, applications can take advantage of thread separation features to split the system into privileged and unprivileged execution environments. Memory protection features provide the capability to partition system resources (memory, peripheral address space, etc) and assign resources to individual threads or groups of threads. Stack, thread execution level, and memory protection constraints are enforced at the time of context switch.

Quality Assurance

The Zephyr project uses an automated quality assurance process. The goal is to have a process including mandatory code reviews, feature and issue management/tracking, and static code analyses.

Code reviews are documented and enforced using a voting system before getting checked into the repository by the responsible subsystem's maintainer. The main goals of the code review are:

- Verifying correct functionality of the implementation
- Increasing the readability and maintainability of the contributed source code
- Ensuring appropriate usage of string and memory functions
- Validation of the user input
- Reviewing the security relevant code for potential issues

The current coding principles focus mostly on coding styles and conventions. Functional correctness is ensured by the build system and the experience of the reviewer. Especially for security relevant code, concrete and detailed guidelines need to be developed and aligned with the developers (see: *Secure Coding*).

Static code analyses are run on the Zephyr code tree on a regular basis using the open source Coverity Scan tool. Coverity Scan now includes complexity analysis.

Bug and issue tracking and management is performed using Jira. The term "survivability" was coined to cover pro-active security tasks such as security issue categorization and management. Initial effort has been started on the definition of vulnerability categorization and mitigation processes within Jira.

Issues determined by Coverity should have more stringent reviews before they are closed as non issues (at least another person educated in security processes need to agree on non-issue before closing).

A security subcommittee has been formed to develop a security process in more detail; this document is part of that process.

Execution Protection

Execution protection is supported and can be categorized into the following tasks:

- **Memory separation:** Memory will be partitioned into regions and assigned attributes based on the owner of that region of memory. Threads will only have access to regions they control.
- **Stack protection:** Stack guards would provide mechanisms for detecting and trapping stack overruns. Individual threads should only have access to their own stacks.
- **Thread separation:** Individual threads should only have access to their own memory resources. As threads are scheduled, only memory resources owned by that thread will be accessible. Topics such as program flow protection and other measures for tamper resistance are currently not in scope.

System Level Security (Ecosystem, . . .)

System level security encompasses a wide variety of categories. Some examples of these would be:

- Secure/trusted boot
- Over the air (OTA) updates
- External Communication
- Device authentication
- Access control of onboard resources
  - Flash updating
  - Secure storage
  - Peripherals
- Root of trust
- Reduction of attack surface

Some of these categories are interconnected and rely on multiple pieces to be in place to produce a full solution for the application.

9.1.3 Secure Development Process

The development of secure code shall adhere to certain criteria. These include coding guidelines and development processes that can be roughly separated into two categories related to software quality and related to software security. Furthermore, a system architecture document shall be created and kept up-to-date with future development.

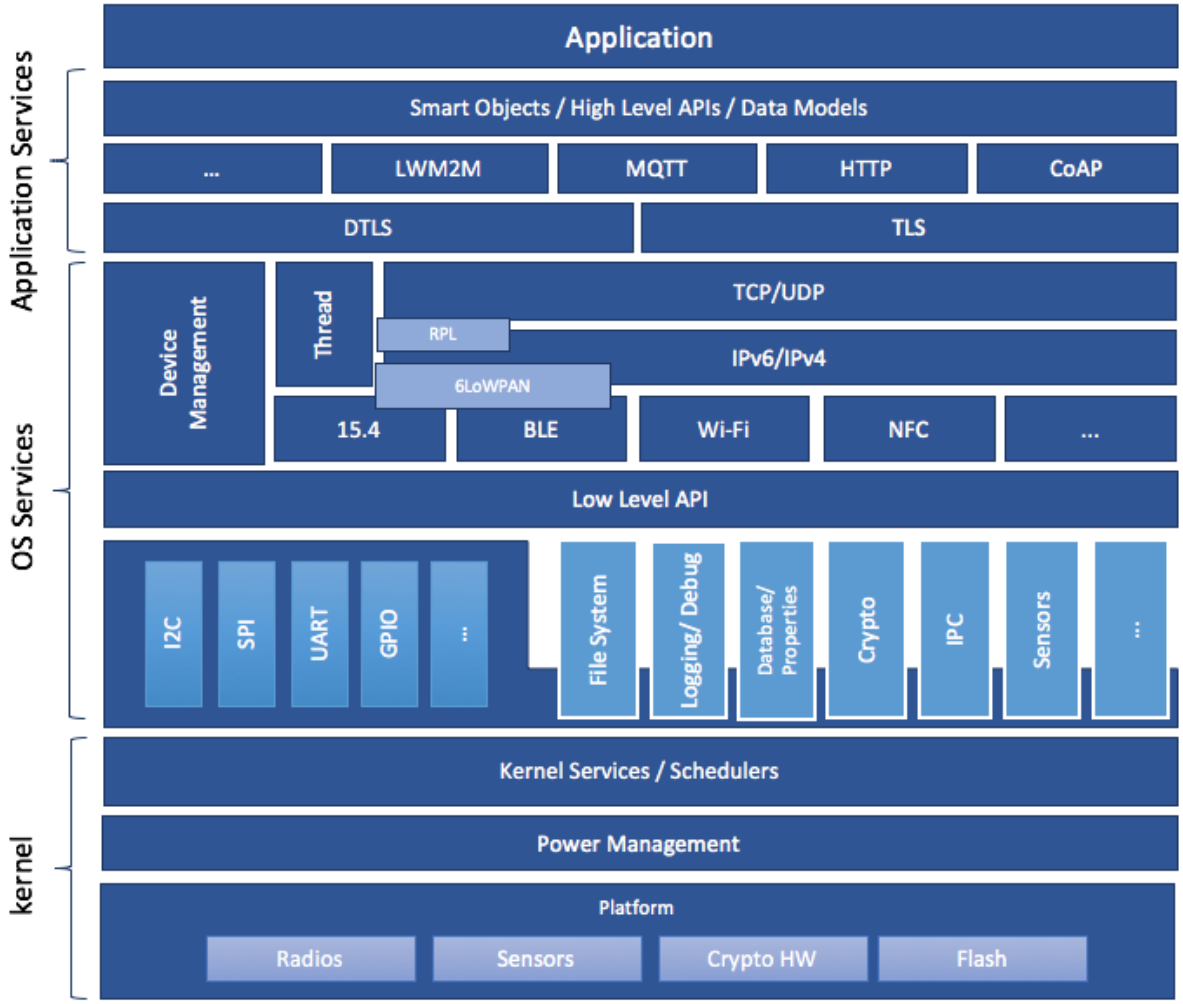


Fig. 2: Figure 2: Zephyr System Architecture

System Architecture

A high-level schematic of the Zephyr system architecture is given in Figure 2. It separates the architecture into an OS part (*kernel + OS Services*) and a user-specific part (*Application Services*). The OS part itself contains low-level, platform specific drivers and the generic implementation of I/O APIs, file systems, kernel-specific functions, and the cryptographic library.

A document describing the system architecture and design choices shall be created and kept up to date with future development. This document shall include the base architecture of the Zephyr OS and an overview of important submodules. For each of the modules, a dedicated architecture document shall be created and evaluated against the implementation. These documents shall serve as an entry point to new developers and as a basis for the security architecture. Please refer to the *Zephyr subsystem documentation* for detailed information.

Secure Coding

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. These standards are included in the Zephyr Project documentation, specifically in its *Secure Coding* section. In [**?**], the following, widely accepted principles for protection mechanisms are defined to prevent security violations and limit their impact:

- **Open design** as a design principle incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [**?**] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the "Secure by Default" paradigm [**?**].
- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [**?**].
- **Least privilege** describes an access model in which each user, program and thread shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.
- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [**?**] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure its usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth:** do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [**?**] paradigm shall be used.
- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely used (a threshold

of 80% is given in [**?**]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.

- **Change management:** to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

Based on these design principles and commonly accepted best practices, a secure development guide shall be developed, published, and implemented into the Zephyr development process. Further details on this are given in the *Secure Design* section.

Quality Assurance

The quality assurance part encompasses the following criteria:

- **Adherence to the Coding Conventions** with respect to coding style, naming schemes of modules, functions, variables, and so forth. This increases the readability of the Zephyr code base and eases the code review. These coding conventions are enforced by automated scripts prior to check-in.
- **Adherence to Deployment Guidelines** is required to ensure consistent releases with a well-documented feature set and a trackable list of security issues.
- **Code Reviews** ensure the functional correctness of the code base and shall be performed on each proposed code change prior to check-in. Code reviews shall be performed by at least one independent reviewer other than the author(s) of the code change. These reviews shall be performed by the subsystem maintainers and developers on a functional level and are to be distinguished from security reviews as laid out in the *Secure Design* section. Refer to the *Development and Contribution Process* documentation for more information.
- **Static Code Analysis** tools efficiently detect common coding mistakes in large code bases. All code shall be analyzed using an appropriate tool prior to merges into the main repository. This is not per individual commit, but is to be run on some interval on specific branches. It is mandatory to remove all findings or waive potential false-positives before each release. Waivers shall be documented centrally and in the form of a comment inside the source code itself. The documentation shall include the employed tool and its version, the date of the analysis, the branch and parent revision number, the reason for the waiver, the author of the respective code, and the approver(s) of the waiver. This shall as a minimum run on the main release branch and on the security branch. It shall be ensured that each release has zero issues with regard to static code analysis (including waivers). Refer to the *Development and Contribution Process* documentation for more information.
- **Complexity Analyses** shall be performed as part of the development process and metrics such as cyclomatic complexity shall be evaluated. The main goal is to keep the code as simple as possible.
- **Automation:** the review process and checks for coding rule adherence are a mandatory part of the precommit checks. To ensure consistent application, they shall be automated as part of the precommit procedure. Prior to merging large pieces of code in from subsystems, in addition to review process and coding rule adherence, all static code analysis must have been run and issues resolved.

Release and Lifecycle Management

Lifecycle management contains several aspects:

- **Device management** encompasses the possibility to update the operating system and/or security related sub-systems of Zephyr enabled devices in the field.

- **Lifecycle management:** system stages shall be defined and documented along with the transactions between the stages in a system state diagram. For security reasons, this shall include locking of the device in case an attack has been detected, and a termination if the end of life is reached.
- **Release management** describes the process of defining the release cycle, documenting releases, and maintaining a record of known vulnerabilities and mitigations. Especially for certification purposes the integrity of the release needs to be ensured in a way that later manipulation (e.g. inserting of backdoors, etc.) can be easily detected.
- **Rights management and NDAs:** if required by the chosen certification, the confidentiality and integrity of the system needs to be ensured by an appropriate rights management (e.g. separate source code repository) and non-disclosure agreements between the relevant parties. In case of a repository shared between several parties, measures shall be taken that no malicious code is checked in.

These points shall be evaluated with respect to their impact on the development process employed for the Zephyr project.

9.1.4 Secure Design

In order to obtain a certifiable system or product, the security process needs to be clearly defined and its application needs to be monitored and driven. This process includes the development of security related modules in all of its stages and the management of reported security issues. Furthermore, threat models need to be created for currently known and future attack vectors, and their impact on the system needs to be investigated and mitigated. Please refer to the *Secure Coding* outlined in the Zephyr project documentation for detailed information.

The software security process includes:

- **Adherence to the Secure Development Coding** is mandatory to avoid that individual components breach the system security and to minimize the vulnerability of individual modules. While this can be partially achieved by automated tests, it is inevitable to investigate the correct implementation of security features such as countermeasures manually in security-critical modules.
- **Security Reviews** shall be performed by a security architect in preparation of each security-targeted release and each time a security-related module of the Zephyr project is changed. This process includes the validation of the effectiveness of implemented security measures, the adherence to the global security strategy and architecture, and the preparation of audits towards a security certification if required.
- **Security Issue Management** encompasses the evaluation of potential system vulnerabilities and their mitigation as described in *Security Issue Management*.

These criteria and tasks need to be integrated into the development process for secure software and shall be automated wherever possible. On system level, and for each security related module of the secure branch of Zephyr, a directly responsible security architect shall be defined to guide the secure development process.

Security Architecture

The general guidelines above shall be accompanied by an architectural security design on system- and module-level. The high level considerations include

- The identification of **security and compliance requirements**
- **Functional security** such as the use of cryptographic functions whenever applicable
- Design of **countermeasures** against known attack vectors
- Recording of security relevant **auditable events**
- Support for **Trusted Platform Modules (TPM)** and **Trusted Execution Environments (TEE)**

- Mechanisms to allow for **in-the-field updates** of devices using Zephyr
- Task scheduler and separation

The security architecture development is based on assets derived from the structural overview of the overall system architecture. Based on this, the individual steps include:

1. **Identification of assets** such as user data, authentication and encryption keys, key generation data (obtained from RNG), security relevant status information.
2. **Identification of threats** against the assets such as breaches of confidentiality, manipulation of user data, etc.
3. **Definition of requirements** regarding security and protection of the assets, e.g. countermeasures or memory protection schemes.

The security architecture shall be harmonized with the existing system architecture and implementation to determine potential deviations and mitigate existing weaknesses. Newly developed sub-modules that are integrated into the secure branch of the Zephyr project shall provide individual documents describing their security architecture. Additionally, their impact on the system level security shall be considered and documented.

Security Vulnerability Reporting

Please see *Security Vulnerability Reporting* for information on reporting security vulnerabilities.

Threat Modeling and Mitigation

The modeling of security threats against the Zephyr RTOS is required for the development of an accurate security architecture and for most certification schemes. The first step of this process is the definition of assets to be protected by the system. The next step then models how these assets are protected by the system and which threats against them are present. After a threat has been identified, a corresponding threat model is created. This model contains the asset and system vulnerabilities, as well as the description of the potential exploits of these vulnerabilities. Additionally, the impact on the asset, the module it resides in, and the overall system is to be estimated. This threat model is then considered in the module and system security architecture and appropriate counter-measures are defined to mitigate the threat or limit the impact of exploits.

In short, the threat modeling process can be separated into these steps (adapted from [**?**]):

1. Definition of assets
2. Application decomposition and creation of appropriate data flow diagrams (DFDs)
3. Threat identification and categorization using the [**?**] and [**?**] approaches
4. Determination of countermeasures and other mitigation approaches

This procedure shall be carried out during the design phase of modules and before major changes of the module or system architecture. Additionally, new models shall be created or existing ones shall be updated whenever new vulnerabilities or exploits are discovered. During security reviews, the threat models and the mitigation techniques shall be evaluated by the responsible security architect.

From these threat models and mitigation techniques tests shall be derived that prove the effectiveness of the countermeasures. These tests shall be integrated into the continuous integration workflow to ensure that the security is not impaired by regressions.

Vulnerability Analyses

In order to find weak spots in the software implementation, vulnerability analyses (VA) shall be performed. Of special interest are investigations on cryptographic algorithms, critical OS tasks, and connectivity protocols.

On a pure software level, this encompasses

- **Penetration testing** of the RTOS on a particular hardware platform, which involves testing the respective Zephyr OS configuration and hardware as one system.
- **Side channel attacks** (timing invariance, power invariance, etc.) should be considered. For instance, ensuring **timing invariance** of the cryptographic algorithms and modules is required to reduce the attack surface. This applies to both the software implementations and when using cryptographic hardware.
- **Fuzzing tests** shall be performed on both exposed APIs and protocols.

The list given above serves primarily illustration purposes. For each module and for the complete Zephyr system (in general on a particular hardware platform), a suitable VA plan shall be created and executed. The findings of these analyses shall be considered in the security issue management process, and learnings shall be formulated as guidelines and incorporated into the secure coding guide.

If possible (as in case of fuzzing analyses), these tests shall be integrated into the continuous integration process.

9.1.5 Security Certification

One goal of creating a secure branch of the Zephyr RTOS is to create a certifiable system or certifiable submodules thereof. The certification scope and scheme is yet to be decided. However, many certification such as Common Criteria [**?**] require evidence that the evaluation claims are indeed fulfilled, so a general certification process is outlined in the following. Based on the final choices for the certification scheme and evaluation level, this process needs to be refined.

Generic Certification Process

In general, the steps towards a certification or precertification (compare [**?**]) are:

1. The **definition of assets** to be protected within the Zephyr RTOS. Potential candidates are confidential information such as cryptographic keys, user data such as communication logs, and potentially IP of the vendor or manufacturer.
2. Developing a **threat model** and **security architecture** to protect the assets against exploits of vulnerabilities of the system. As a complete threat model includes the overall product including the hardware platform, this might be realized by a split model containing a precertified secure branch of Zephyr which the vendor could use to certify their Zephyr-enabled product.
3. Formulating an **evaluation target** that includes the **certification claims** on the security of the assets to be evaluated and certified, as well as assumptions on the operating conditions.
4. Providing **proof** that the claims are fulfilled. This includes consistent documentation of the security development process, etc.

These steps are partially covered in previous sections as well. In contrast to these sections, the certification process only requires to consider those components that shall be covered by the certification. The security architecture, for example, considers assets on system level and might include items not relevant for the certification.

Certification Options

For the security certification as such, the following options can be pursued:

1. **Abstract precertification of Zephyr as a pure software system:** this option requires assumptions on the underlying hardware platform and the final application running on top of Zephyr. If these assumptions are met by the hardware and the application, a full certification can be more easily achieved. This option is the most flexible approach but puts the largest burden on the product vendor.

2. **Certification of Zephyr on specific hardware platform without a specific application in mind:** this scenario describes the enablement of a secure platform running the Zephyr RTOS. The hardware manufacturer certifies the platform under defined assumptions on the application. If these are met, the final product can be certified with little effort.
3. **Certification of an actual product:** in this case, a full product including a specific hardware, the Zephyr RTOS, and an application is certified.

In all three cases, the certification scheme (e.g. FIPS 140-2 [**?**] or Common Criteria [**?**]), the scope of the certification (main-stream Zephyr, security branch, or certain modules), and the certification/assurance level need to be determined.

In case of partial certifications (options 1 and 2), assumptions on hardware and/or software are required for certifications. These can include [**?**]

- **Appropriate physical security** of the hardware platform and its environment.
- **Sufficient protection of storage and timing channels** on the hardware platform itself and all connected devices. (No mentioning of remote connections.)
- Only **trusted/assured applications** running on the device
- The device and its software stack is configured and operated by **properly trained and trusted individuals** with no malicious intent.

These assumptions shall be part of the security claim and evaluation target documents.

9.2 Security Vulnerability Reporting

9.2.1 Introduction

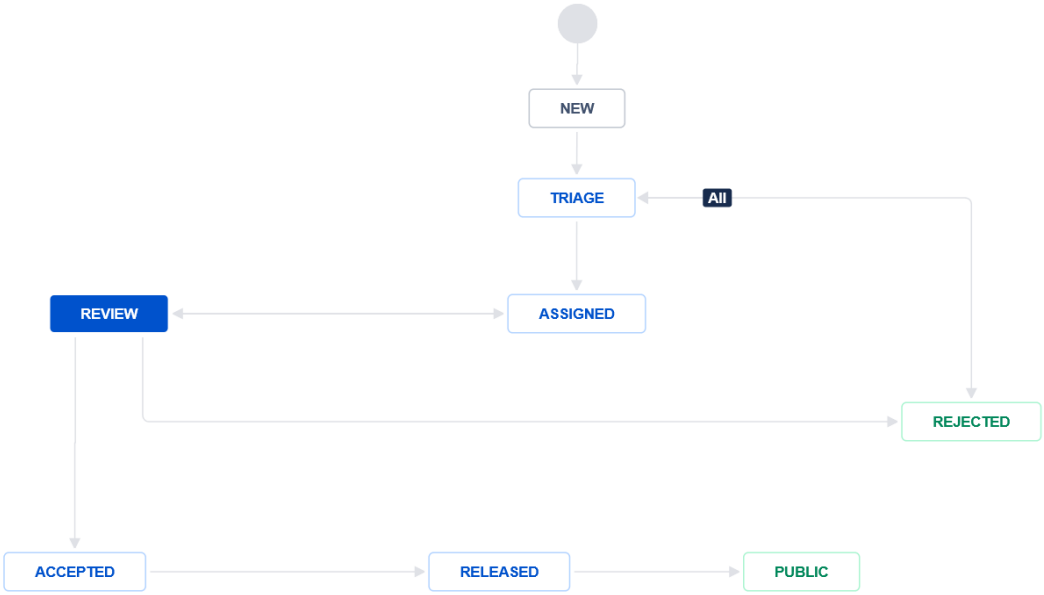
Vulnerabilities to the Zephyr project may be reported via email to the vulnerabilities@zephyrproject.org mailing list. These reports will be acknowledged and analyzed by the security response team within 1 week. Each vulnerability will be entered into the Zephyr Project security tracking JIRA. The original submitter will be granted permission to view the issues that they have reported.

Reporters may also submit reports by directly submitting them to the Zephyr Product security tracking JIRA.

9.2.2 Security Issue Management

Issues within this bug tracking system will transition through a number of states according to this diagram:

- New: This state represents new reports that have been entered directly by a reporter. When entered by the response team in response to an email, the issue shall be transitioned directly to Triage.
- Triage: This issue is awaiting Triage by the response team. The response team will analyze the issue, determine a responsible entity, assign the JIRA ticket to that individual, and move the issue to the Assigned state. Part of triage will be to set the issue's priority.
- Assigned: The issue has been assigned, and is awaiting a fix by the assignee.
- Review: Once there is a Zephyr pull request for the issue, the PR link will be added to a comment in the issue, and the issue moved to the Review state.
- Accepted: Indicates that this issue has been merged into the appropriate branch within Zephyr.
- Release: The PR has been included in a released version of Zephyr.
- Public: The embargo period has ended. The issue will be made publicly visible, the associated CVE updated, and the vulnerabilities page in the docs updated to include the detailed information.



The issues created in this JIRA instance are kept private, due to the sensitive nature of security reports. The issues are only visible to certain parties:

- Members of the PSIRT mailing list
- the reporter
- others, as proposed and ratified by the Zephyr Security Subcommittee. In the general case, this will include:
  - The code owner responsible for the fix.
  - The Zephyr release owners for the relevant releases affected by this vulnerability.

The Zephyr Security Subcommittee shall review the reported vulnerabilities during any meeting with more than three people in attendance. During this review, they shall determine if new issues need to be embargoed.

The guideline for embargo will be based on: 1. Severity of the issue, and 2. Exploitability of the issue. Issues that the subcommittee decides do not need an embargo will be reproduced in the regular Zephyr project bug tracking system, and a comment added to the JIRA issue pointing to the bug tracking issue. These issues will be marked as being tracked within the Zephyr bug tracking system.

Security sensitive vulnerabilities shall be made public after an embargo period of at most 90 days. The intent is to allow 30 days within the Zephyr project to fix the issues, and 60 days for external parties building products using Zephyr to be able to apply and distribute these fixes.

Fixes to the code shall be made through pull requests PR in the Zephyr project github. Developers shall make an attempt to not reveal the sensitive nature of what is being fixed, and shall not refer to CVE numbers that have been assigned to the issue. The developer instead should merely describe what has been fixed.

The security subcommittee will maintain information mapping embargoed CVEs to these PRs (this information is within the JIRA issues), and produce regular reports of the state of security issues.

Each JIRA issue that is considered a security vulnerability shall be assigned a CVE number. As fixes are created, it may be necessary to allocate additional CVE numbers, or to retire numbers that were assigned.

9.2.3 Vulnerability Notification

Each Zephyr release shall contain a report of CVEs that were fixed in that release. Because of the sensitive nature of these vulnerabilities, the release shall merely include a list of CVEs that have been fixed. After the embargo period, the vulnerabilities page shall be updated to include additional details of these vulnerabilities. The vulnerability page shall give credit to the reporter(s) unless a reporter specifically requests anonymity.

The Zephyr project shall maintain a vulnerability-alerts mailing list. This list will be seeded initially with a contact from each project member. Additional parties can request to join this list by filling out the form at the Vulnerability Registry. These parties will be vetted by the project director to determine that they have a legimitate interest in knowing about security vulnerabilities during the embargo period.

Periodically, the security subcommittee will send information to this mailing list describing known embargoed issues, and their backport status within the project. This information is intended to allow them to determine if they need to backport these changes to any internal trees.

When issues have been triaged, this list will be informed of:

- The Zephyr Project security JIRA link (ZEPSEC).
- The CVE number assigned.
- The subsystem involved.
- The severity of the issue.

After acceptance of a PR fixing the issue (merged), in addition to the above, the list will be informed of:

- The association between the CVE number and the PR fixing it.
- Backport plans within the Zephyr project.

9.2.4 Backporting of Security Vulnerabilities

Each security issue fixed within zephyr shall be backported to the following releases:

- The current Long Term Stable (LTS) release.
- The most recent two releases.

The developer of the fix shall be responsible for any necessary backports, and apply them to any of the above listed release branches, unless the fix does not apply (the vulnerability was introduced after this release was made).

Backports will be tracked on the security JIRA instance using a subtask issue of type “backport”.

9.2.5 Need to Know

Due to the sensitive nature of security vulnerabilities, it is important to share details and fixes only with those parties that have a need to know. The following parties will need to know details about security vulnerabilities before the embargo period ends:

- Maintainers will have access to all information within their domain area only.
- The current release manager, and the release manager for historical releases affected by the vulnerability (see backporting above).
- The Project Security Incident Response (PSIRT) team will have full access to information. The PSIRT is made up of representatives from platinum members, and volunteers who do work on triage from other members.
- As needed, release managers and maintainers may be invited to attend additional security meetings to discuss vulnerabilities.

9.3 Secure Coding

Traditionally, microcontroller-based systems have not placed much emphasis on security. They have usually been thought of as isolated, disconnected from the world, and not very vulnerable, just because of the difficulty in accessing them. The Internet of Things has changed this. Now, code running on small microcontrollers often has access to the internet, or at least to other devices (that may themselves have vulnerabilities). Given the volume they are often deployed at, uncontrolled access can be devastating[1].

This document describes the requirements and process for ensuring security is addressed within the Zephyr project. All code submitted should comply with these principles.

Much of this document comes from [**?**].

9.3.1 Introduction and Scope

This document covers guidelines for the Zephyr Project, from a security perspective. Many of the ideas contained herein are captured from other open source efforts.

We begin with an overview of secure design as it relates to Zephyr. This is followed by a section on *Secure development knowledge*, which gives basic requirements that a developer working on the project will need to have. This section gives references to other security documents, and full details of how to write secure software are beyond the scope of this document. This section also describes vulnerability knowledge that at least one of the primary developers should have. This knowledge will be necessary for the review process described below this.

Following this is a description of the review process used to incorporate changes into the Zephyr codebase. This is followed by documentation about how security-sensitive issues are handled by the project.

Finally, the document covers how changes are to be made to this document.

9.3.2 Secure Coding

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. In [**?**], the following, widely accepted principles for protection mechanisms are defined to help prevent security violations and limit their impact:

- **Open design** as a design guideline incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [**?**] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the "Secure by Default" paradigm [**?**].
- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [**?**].
- **Least privilege** describes an access model in which each user, program, and thread, shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.

[1] An attack resulted in a significant portion of DNS infrastructure being taken down.

- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [**?**] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure their usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth**: do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [**?**] paradigm shall be used.
- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely used (a threshold of 80% is given in [**?**]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.
- **Change management**: to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

9.3.3 Secure development knowledge

Secure designer

The Zephyr project must have at least one primary developer who knows how to design secure software.

This requires understanding the following design principles, including the 8 principles from [**?**]:

- economy of mechanism (keep the design as simple and small as practical, e.g., by adopting sweeping simplifications)
- fail-safe defaults (access decisions shall deny by default, and projects' installation shall be secure by default)
- complete mediation (every access that might be limited must be checked for authority and be non-bypassable)
- open design (security mechanisms should not depend on attacker ignorance of its design, but instead on more easily protected and changed information like keys and passwords)
- separation of privilege (ideally, access to important objects should depend on more than one condition, so that defeating one protection system won't enable complete access. For example, multi-factor authentication, such as requiring both a password and a hardware token, is stronger than single-factor authentication)
- least privilege (processes should operate with the least privilege necessary)
- least common mechanism (the design should minimize the mechanisms common to more than one user and depended on by all users, e.g., directories for temporary files)
- psychological acceptability (the human interface must be designed for ease of use - designing for "least astonishment" can help)

- limited attack surface (the set of the different points where an attacker can try to enter or extract data)
- input validation with whitelists (inputs should typically be checked to determine if they are valid before they are accepted; this validation should use whitelists (which only accept known-good values), not blacklists (which attempt to list known-bad values)).

Vulnerability Knowledge

A "primary developer" in a project is anyone who is familiar with the project's code base, is comfortable making changes to it, and is acknowledged as such by most other participants in the project. A primary developer would typically make a number of contributions over the past year (via code, documentation, or answering questions). Developers would typically be considered primary developers if they initiated the project (and have not left the project more than three years ago), have the option of receiving information on a private vulnerability reporting channel (if there is one), can accept commits on behalf of the project, or perform final releases of the project software. If there is only one developer, that individual is the primary developer.

At least one of the primary developers **must** know of common kinds of errors that lead to vulnerabilities in this kind of software, as well as at least one method to counter or mitigate each of them.

Examples (depending on the type of software) include SQL injection, OS injection, classic buffer overflow, cross-site scripting, missing authentication, and missing authorization. See the CWE/SANS top 25 or OWASP Top 10 for commonly used lists.

Zephyr Security Subcommittee

There shall be a "Zephyr Security Subcommittee", responsible for enforcing this guideline, monitoring reviews, and improving these guidelines.

This team will be established according to the Zephyr Project charter.

9.3.4 Code Review

The Zephyr project shall use a code review system that all changes are required to go through. Each change shall be reviewed by at least one primary developer that is not the author of the change. This developer shall determine if this change affects the security of the system (based on their general understanding of security), and if so, shall request the developer with vulnerability knowledge, or the secure designer to also review the code. Any of these individuals shall have the ability to block the change from being merged into the mainline code until the security issues have been addressed.

9.3.5 Issues and Bug Tracking

The Zephyr project shall have an issue tracking system (such as JIRA) that can be used to record and track defects that are found in the system.

Because security issues are often sensitive, this issue tracking system shall have a field to indicate a security issue. Setting this field shall result in the issue only being visible to the Zephyr Security Subcommittee. In addition, there shall be a field to allow the Zephyr Security Subcommittee to add additional users that will have visibility to a given issue.

This embargo, or limited visibility, shall only be for a fixed duration, with a default being a project-decided value. However, because security considerations are often external to the Zephyr project itself, it may be necessary to increase this embargo time. The time necessary shall be clearly annotated in the issue itself.

The list of issues shall be reviewed at least once a month by the Zephyr Security Subcommittee. This review should focus on tracking the fixes, determining if any external parties need to be notified or

involved, and determining when to lift the embargo on the issue. The embargo should **not** be lifted via an automated means, but the review team should avoid unnecessary delay in lifting issues that have been resolved.

9.3.6 Modifications to This Document

Changes to this document shall be reviewed by the Zephyr Security Subcommittee, and approved by consensus.

9.4 Sensor Device Threat Model

This document describes a threat model for an IoT sensor device. Spelling out a threat model helps direct development effort, and can be used to help prioritize these efforts as well.

This device contains a sensor of some type (for example temperature, or a pressure in a pipe), which sends this data to an SoC running a microcontroller. This microcontroller connects to a cloud service, and relays this sensor data to this service. The cloud service is also able to send configuration data to the device, as well as software update images. A general diagram can be seen in Figure 1:

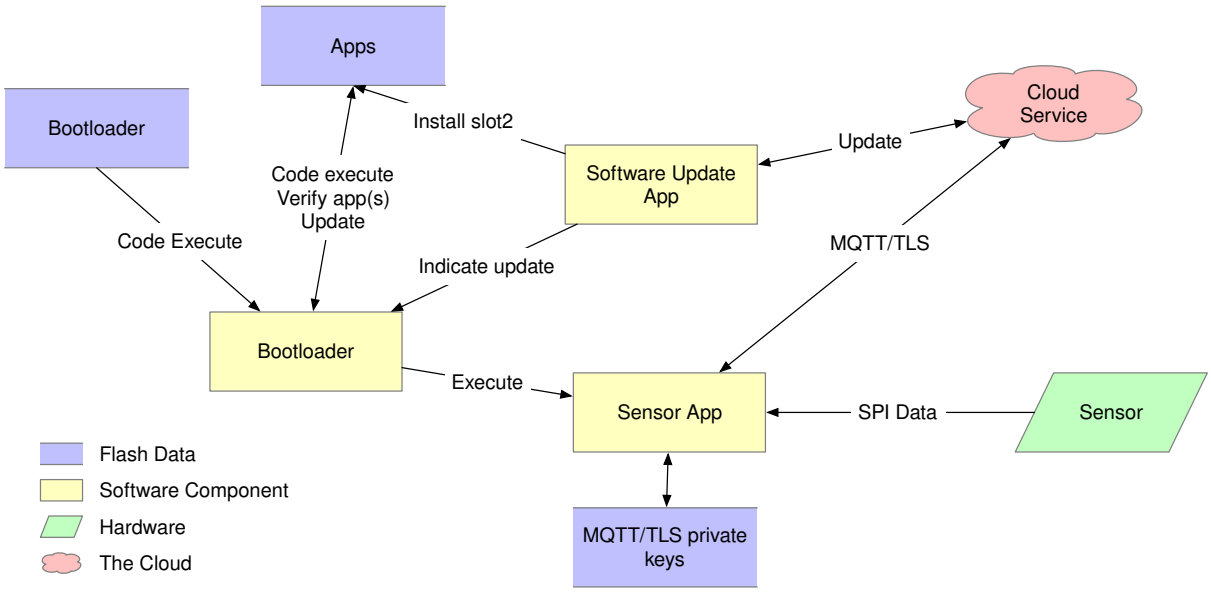


Fig. 3: Figure 1. Sensor General Diagram

In this sensor device, the sensor connects with the SoC via an SPI bus, and the SoC has a network interface that it uses to communicate with the cloud service. The particulars of these interfaces can impact the threat model in unexpected ways, and variants on this will need to be considered (for example, using a separate network interface SoC connected via some type of bus).

This model also focuses on communicating via the MQTT-over-TLS protocol, as this seems to be in wide use[1].

9.4.1 Assets

One aspect of the threat model to consider are assets involved in the operation of the device. The following list enumerates the assets included in this model:

[1] See https://www.slideshare.net/kartben/iot-developer-survey-2018. As of this writing, the three major cloud IoT service providers, AWS IoT, Google Cloud IoT, and Microsoft Azure IoT all provide MQTT over TLS. Some feedback has suggested that some find difficulty with UDP protocols and routing issues on various networks.

1. **The bootloader**. This is a small code/data image contained in on-device flash that is the first code to run. In order to establish a root of trust, this image must be immutable. This model assumes that the SoC provides a mechanism to protect a region of the flash from future writes, and that this will be done after this image is programmed into the device, early in production [th-imboot].

2. **The application firmware image**. This asset consists of the remainder of the firmware run by the microcontroller. The distinction is made because this part of the image will need to be updated periodically as security vulnerabilities are discovered. Requirements for updates to this image are:

   a. The image shall only be replaced with an authorized image [th-authrepl].

   b. When an authorized replacement image is available, the update shall be done in a timely manner [th-timely-update].

   c. The image update shall be seen as atomic, meaning that when the image is run, the flash shall contain either the update image in its entirety, or the old image in its entirety [th-atomic-update].

3. **Root certificate list**. In order to authenticate the cloud service (server), the IoT device must have a list of root certificates that are allowed to sign the certificate on the server. For cloud-provider based services, this list will generally be provided by the service provider. Because the root certificates can expire, and possibly be revoked, this list will need to be periodically updated [th-root-certs], [th-root-check].

4. **Client secrets**. To authenticate the client to the service, the client must possess some kind of secret. This is generally a private key, usually either an RSA key or an EC private key. When establishing communication with the server, the device will use this secret either as part of the TLS establishment, or to sign a message used in the communication.

   This secret is generally generated by the service provider, or by software running elsewhere, and must be securely installed on the device. Policy may dictate that this secret be replaced periodically, which will require a way to update the client secret. Typically, the service will allow two or three active keys to allow this update to proceed while the old key is used.

   These secrets must be protected from read, and the smallest amount of code necessary shall have access to them. [th-secret-storage]

5. **Current date/time**. TLS certificate verification requires knowledge of the current date and time in order to determine if the current time falls within the certificate's current validity time. Also, token based client authentication will generally require the client to sign a message containing a time window that the token is valid. Certificate validation requires the device's notion of date and time to be accurate within a day or so. Token generation generally requires the time to be accurate within 5-10 minutes.

   It may be possible to approximate secure time by querying an external time server. Secure NTP is possibly beyond the capabilities of an IoT device. The main risks of having incorrect time are denial of service (the device rejects valid certificates), and the generation of tokens with invalid times. It could be possible to trick the device into generating tokens that are valid in the future, but the attacker would also have to spoof the server's certificate to be able to intercept this. [th-time]

6. **Sensor data**. The data received from the sensor itself, and delivered to the service shall be delivered without modification or tampering.

7. **Device configuration**. Various configuration data, such as the hostname of the service to connect to, the address of a time server, frequency and parameters of when sensor data is sent to the service, and other need to be kept by the device. This configuration data will need to be updated periodically as the configuration changes. Updates should be allowed only from authorized parties. [th-conf]

8. **Logs**. In order to assist with analysis of security issues, the device shall log information about security-pertinent events. IoT devices generally have limited storage, and as such, these logs need to be carefully selected. It may also be possible to send these log events to the cloud service where they can be stored in a more resource-available environment. Types of events that should be logged include:

a. **Firmware image updates**. The system should log the download of new images, and when an image is successfully updated.

b. **Client secret changes**. Changes and new client secrets should be logged.

c. **Changes to the device configuration**.

[th-logs]

9.4.2 Communication

In addition to assets, the threat model also considers the locations where data or assets are communicated between entities of the system.

1. **Flash contents**. The flash device contains several regions. The contents of flash can be modified programmatically by the SoC's CPU.

   a. **The bootloader**. As described in the Assets section, the bootloader is a small section of the flash device containing the code initially run. This section shall be written early in the lifecycle of the device, and the flash device then configured to permanently disallow modification of this section. This configuration should also prevent modification via external interfaces, such as JTAG or SWD debuggers.

      The bootloader is responsible for verifying the signature of the application image as well as updating the application image from the update image when an update is needed.

      The bootloader shall verify the signature of the update image before installing it.

      The bootloader shall only accept an update image with a newer version number than the current image.

   b. **The application image**. The application image contains the code executed during normal operation of the device. Before running this image, the bootloader shall verify a digital signature of the image, to avoid running an image that has been tampered with. The flash/system shall be configured such that after the bootloader has completed, the CPU will be unable to write to the application image.

   c. **The update image**. This is an area of flash that holds a new version of the application image. This image will be downloaded and stored by the application during normal operation. When this has completed, the application can trigger a reboot, and the bootloader can install the new image.

   d. **Secret storage**. An area of the flash will be used to store client secrets. This area is written and read by a subset of the application image. The application shall be configured to protect this area from both reads and writes by code that does not need to have access to it, giving consideration to possible exploits found within a majority of the application code. Revealing the contents of the secrets would allow the attacker to spoof this device.

      Initial secrets shall be placed in the device during a provisioning activity, distinct from normal operation of the device. Later updates can be made under the direction of communication received over a secured channel to the service.

   e. **Configuration storage**. There shall be an area to store other configuration information. On resource-constrained devices, it is allowed for this to be stored in the same region as the secret storage, however, this adds additional code that has access to the secret storage area, and as such, more code that must be scrutinized.

   f. **Log storage**. The device may have an area of flash where log events can be written.

2. **Sensor/Actuator interface**. In this design, the sensor or actuator communicates with the SoC via a bus, such as SPI. The hardware design shall be made to make intercepting this bus difficult for an attack. Required techniques depend on the sensitivity and use of the sensor data, and can range from having the sensor mounted on the same PCB as the MCU to epoxy potting the entire device.

3. **Communication with cloud service**. Communication between the device, and the cloud service will be done over the general internet. As such, it shall be assumed that an attacker can arbitrarily intercept this channel and, for example, return spoofed DNS results or attempt man-in-the-middle attacks against communication with cloud services.

   The device shall use TLS for all communication with the cloud service [th-all-tls]. The TLS stack shall be configured to use only cipher suites that are generally considered secure[2], including forward secrecy. The communication shall be secured by the following:

   a. **Cipher suite selection**. The device shall only allow communication with generally agreed secure cipher suites [th-tls-ciphers].

   b. **Server certificate verification**. The server presented by the server shall be verified [th-root-check].

      i. **Naming**. The certificate shall name the host and service the cloud service server is providing. RFC6125 describes best practices for this. It is permissible for the device to require the certificate to be more restrictive than as described in this RFC, provided the service can use a certificate that can comply.

      ii. **Path validation**. The device shall verify that the certificate chain has a valid signature path from a root certificate contained within the device, to the certificate presented by the service. RFC4158 describes this is general. The device is permitted to require a more restricted path, provided the server certificate used complies with this restriction.

      iii. **Validity period**. The validity period of all presented certificates shall be checked against the device's best notion of the current time.

   c. **Client authentication**. The client shall authenticate itself to the service using a secret known only to that particular device. There are several options, and the technique used is generally mandated by the particular service provider being used [th-tls-client-auth].

      i. **TLS client certificates**. The TLS protocol allows the client to present a certificate, and assert its knowledge of the secret described by that certificate. Generally, these certificates will be stored within the service provider. These certificates can be self-signed, or signed by a CA. Since the service provider maintains a list of valid certificates (mapping them to a device identity), having these certificates signed by a CA does not add any additional security, but may be useful in the management of these certificates.

      ii. **Token-based authentication**. It is also possible for the client to authenticate itself using the *password* field of the MQTT CONNECT packet. However, the secret itself must not be transmitted in this packet. Instead, a token-based protocol, such as RFC7519's JSON Web Token (JWT) can be used. These tokens will generally have a small validity period (e.g. 1 hour), to prevent them from being reused if they are intercepted. The token shall not be sent until the device has verified the identity of the server.

   d. **Random/Entropy source**. Cryptograph communication requires the generation of secure pseudorandom numbers. The device shall use a modern, accepted cryptographic random-bit generator to generate these random numbers. It shall use either a Non-Deterministic Random Bit Generator (True RBG) implemented in hardware within the SoC, or a Deterministic Random Bit Generator (Pseudo RBG) seeded by an entropy source within the SoC. Please see NIST SP 800-90A for information on approved RBGs and NIST SP 800-90B for information on testing a device's entropy source [th-entropy].

4. **Communication with the time service**. Ideally, the device shall contain hardware that maintains a secure time. However, most SoCs in use do not have support for this, and it will be necessary to consult an external time service. RFC4330 and referenced RFCs describe the Simple Network Time Protocol that can be used to query the current time from a network time server.

5. **Device lifecycle**. An IoT device will have a lifecycle from production to destruction and disposal of the device. Aspects of this lifecycle that impact security include initial provisioning, normal operation, re-provisioning, and destruction.

[2] As new exploits are discovered, what is considered secure can change. Organizations such as https://www.ssllabs.com/ provide information on current ideas of how TLS must be configured to be secure.

a. **Initial provisioning**. During the initial provisioning stage, it is necessary to program the bootloader, an initial application image, a device secret, and initial configuration data [th-initial-provision]. In addition, the bootloader flash protection shall be installed. Of this information, only the device secret needs to differ per device. This secret shall be securely maintained, and destroyed in all locations outside of the device once it has been programmed [th-initial-secret].

b. **Normal operation**. Normal operation includes the behavior described by the rest of this document.

c. **Re-provisioning**. Sometimes it is necessary to re-provision a device, such as for a different application. One way to do this is to keep the same device secret, and replace the configuration data, as well as the cloud service data associated with the device. It is also possible to program a new device secret, but if this is done it shall be done securely, and the new secret destroyed externally once programmed into the device [th-reprovision].

d. **Destruction**. To prevent the device secret from being used to spoof the device, upon decommissioning, the secret for a particular device shall be rendered ineffective [th-destruction]. Possibilities include:

   i. Hardware destruction of the device.

   ii. Securely wiping the flash area containing the secret[3].

   iii. Removing the device identity and certificate from the service.

9.4.3 Other Considerations

In addition to the above, network connected devices generally will need a way to configure them to connect to the network environment they are placed in. There are numerous ways of doing this, and it is important for these configuration methods to not circumvent the security requirements described above.

9.4.4 Threats

9.4.5 Notes

9.5 Hardening Tool

Zephyr contains several optional features that make the overall system more secure. As we take advantage of hardware features, many of these options are platform specific and besides it, some of them are unknown by developers.

To address this problem, Zephyr provides a tool that helps to check an application configuration option list against a list of hardening preferences defined by the **Security Group**. The tool can identify the build target and based on that provides suggestions and recommendations on how to optimize the configuration for security.

9.5.1 Usage

After configure of your application, change directory to the build folder and:

```
# ninja build system:
$ ninja hardenconfig
# make build system:
$ make hardenconfig
```

[3] Note that merely erasing this flash area is unlikely to be sufficient.

The output should be similar to the one bellow:

```
                      name                      |   current   |    recommended    ␣
↪||        check result
================================================================================================================
CONFIG_HW_STACK_PROTECTION                      |      n      |         y         ␣
↪||            FAIL
CONFIG_BOOT_BANNER                              |      y      |         n         ␣
↪||            FAIL
CONFIG_PRINTK                                   |      y      |         n         ␣
↪||            FAIL
CONFIG_EARLY_CONSOLE                            |      y      |         n         ␣
↪||            FAIL
CONFIG_OVERRIDE_FRAME_POINTER_DEFAULT           |      n      |         y         ␣
↪||            FAIL
CONFIG_DEBUG_INFO                               |      y      |         n         ␣
↪||            FAIL
CONFIG_TEST_RANDOM_GENERATOR                    |      y      |         n         ␣
↪||            FAIL
CONFIG_BUILD_OUTPUT_STRIPPED                    |      n      |         y         ␣
↪||            FAIL
CONFIG_STACK_SENTINEL                           |      n      |         y         ␣
↪||            FAIL
```

9.6 Vulnerabilities

This page collects all of the vulnerabilities that are discovered and fixed in each release. It will also often have more details than is available in the releases. Some vulnerabilities are deemed to be sensitive, and will not be publicly discussed until there is sufficient time to fix them. Because the release notes are locked to a version, the information here can be updated after the embargo is lifted.

9.6.1 CVE-2017

CVE-2017-14199

Buffer overflow in `getaddrinfo()`.

- CVE-2017-14199
- Zephyr project bug tracker ZEPSEC-12
- PR6158 fix for 1.11.0

CVE-2017-14201

The shell DNS command can cause unpredictable results due to misuse of stack variables.

Use After Free vulnerability in the Zephyr shell allows a serial or telnet connected user to cause denial of service, and possibly remote code execution.

This has been fixed in release v1.14.0.

- CVE-2017-14201
- Zephyr project bug tracker ZEPSEC-17
- PR13260 fix for v1.14.0

CVE-2017-14202

The shell implementation does not protect against buffer overruns resulting in unpredictable behavior.

Improper Restriction of Operations within the Bounds of a Memory Buffer vulnerability in the shell component of Zephyr allows a serial or telnet connected user to cause a crash, possibly with arbitrary code execution.

This has been fixed in release v1.14.0.

- CVE-2017-14202
- Zephyr project bug tracker ZEPSEC-18
- PR13048 fix for v1.14.0

9.6.2 CVE-2019

CVE-2019-9506

The Bluetooth BR/EDR specification up to and including version 5.1 permits sufficiently low encryption key length and does not prevent an attacker from influencing the key length negotiation. This allows practical brute-force attacks (aka "KNOB") that can decrypt traffic and inject arbitrary ciphertext without the victim noticing.

- CVE-2019-9506
- Zephyr project bug tracker ZEPSEC-20
- PR18702 fix for v1.14.0
- PR18659 fix for v2.0.0

9.6.3 CVE-2020

CVE-2020-10019

Buffer Overflow vulnerability in USB DFU of zephyr allows a USB connected host to cause possible remote code execution.

This has been fixed in releases v1.14.2, v2.2.0, and v2.1.1.

- CVE-2020-10019
- Zephyr project bug tracker ZEPSEC-25
- PR23460 fix for 1.14.x
- PR23457 fix for 2.1.x
- PR23190 fix in 2.2.0

CVE-2020-10021

Out-of-bounds write in USB Mass Storage with unaligned sizes

Out-of-bounds Write in the USB Mass Storage memoryWrite handler with unaligned Sizes.

See NCC-ZEP-024, NCC-ZEP-025, NCC-ZEP-026

This has been fixed in releases v1.14.2, and v2.2.0.

- CVE-2020-10021
- Zephyr project bug tracker ZEPSEC-26

- PR23455 fix for v1.14.2
- PR23456 fix for the v2.1 branch
- PR23240 fix for v2.2.0

CVE-2020-10022

UpdateHub Module Copies a Variable-Size Hash String Into a Fixed-Size Array

A malformed JSON payload that is received from an UpdateHub server may trigger memory corruption in the Zephyr OS. This could result in a denial of service in the best case, or code execution in the worst case.

See NCC-ZEP-016

This has been fixed in the below pull requests for main, branch from v2.1.0, and branch from v2.2.0.

- CVE-2020-10022
- Zephyr project bug tracker ZEPSEC-28
- PR24154 fix for main
- PR24065 fix for branch from v2.1.0
- PR24066 fix for branch from v2.2.0

CVE-2020-10023

Shell Subsystem Contains a Buffer Overflow Vulnerability In shell_spaces_trim

The shell subsystem contains a buffer overflow, whereby an adversary with physical access to the device is able to cause a memory corruption, resulting in denial of service or possibly code execution within the Zephyr kernel.

See NCC-ZEP-019

This has been fixed in releases v1.14.2, v2.2.0, and in a branch from v2.1.0,

- CVE-2020-10023
- Zephyr project bug tracker ZEPSEC-29
- PR23646 fix for v1.14.2
- PR23649 fix for branch from v2.1.0
- PR23304 fix for v2.2.0

CVE-2020-10024

ARM Platform Uses Signed Integer Comparison When Validating Syscall Numbers

The arm platform-specific code uses a signed integer comparison when validating system call numbers. An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

See NCC-ZEP-001

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0,

- CVE-2020-10024
- Zephyr project bug tracker ZEPSEC-30
- PR23535 fix for v1.14.2

- PR23498 fix for branch from v2.1.0
- PR23323 fix for v2.2.0

CVE-2020-10027

ARC Platform Uses Signed Integer Comparison When Validating Syscall Numbers

An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

See NCC-ZEP-001

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0.

- CVE-2020-10027
- Zephyr project bug tracker ZEPSEC-35
- PR23500 fix for v1.14.2
- PR23499 fix for branch from v2.1.0
- PR23328 fix for v2.2.0

CVE-2020-10028

Multiple Syscalls In GPIO Subsystem Performs No Argument Validation

Multiple syscalls with insufficient argument validation

See NCC-ZEP-006

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0.

- CVE-2020-10028
- Zephyr project bug tracker ZEPSEC-32
- PR23733 fix for v1.14.2
- PR23737 fix for branch from v2.1.0
- PR23308 fix for v2.2.0 (gpio patch)

CVE-2020-10058

Multiple Syscalls In kscan Subsystem Performs No Argument Validation

Multiple syscalls in the Kscan subsystem perform insufficient argument validation, allowing code executing in userspace to potentially gain elevated privileges.

See NCC-ZEP-006

This has been fixed in a branch from v2.1.0, and release v2.2.0.

- CVE-2020-10058
- Zephyr project bug tracker ZEPSEC-34
- PR23748 fix for branch from v2.1.0
- PR23308 fix for v2.2.0 (kscan patch)

CVE-2020-10059

UpdateHub Module Explicitly Disables TLS Verification

The UpdateHub module disables DTLS peer checking, which allows for a man in the middle attack. This is mitigated by firmware images requiring valid signatures. However, there is no benefit to using DTLS without the peer checking.

See NCC-ZEP-018

This has been fixed in a PR against Zephyr main.

- CVE-2020-10059
- Zephyr project bug tracker ZEPSEC-36
- PR24954 fix on main (to be fixed in v2.3.0)
- PR24954 fix v2.1.0
- PR24954 fix v2.2.0

CVE-2020-10060

UpdateHub Might Dereference An Uninitialized Pointer

In updatehub_probe, right after JSON parsing is complete, objects[1] is accessed from the output structure in two different places. If the JSON contained less than two elements, this access would reference unitialized stack memory. This could result in a crash, denial of service, or possibly an information leak.

Recommend disabling updatehub until such a time as a fix can be made available.

See NCC-ZEP-030

This has been fixed in a PR against Zephyr main.

- CVE-2020-10060
- Zephyr project bug tracker ZEPSEC-37
- PR27865 fix on main (to be fixed in v2.4.0)
- PR27865 fix for v2.3.0
- PR27865 fix for v2.2.0
- PR27865 fix for v2.1.0

CVE-2020-10061

Error handling invalid packet sequence

Improper handling of the full-buffer case in the Zephyr Bluetooth implementation can result in memory corruption.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- CVE-2020-10061
- Zephyr project bug tracker ZEPSEC-75
- PR23516 fix for v2.3 (split driver)
- PR23517 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23547 fix for branch from v2.2.0

CVE-2020-10062

Packet length decoding error in MQTT

CVE: An off-by-one error in the Zephyr project MQTT packet length decoder can result in memory corruption and possible remote code execution. NCC-ZEP-031

The MQTT packet header length can be 1 to 4 bytes. An off-by-one error in the code can result in this being interpreted as 5 bytes, which can cause an integer overflow, resulting in memory corruption.

This has been fixed in main for v2.3.

- CVE-2020-10062
- Zephyr project bug tracker ZEPSEC-84
- commit 11b7a37d for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE-2020-10063

Remote Denial of Service in CoAP Option Parsing Due To Integer Overflow

A remote adversary with the ability to send arbitrary CoAP packets to be parsed by Zephyr is able to cause a denial of service.

This has been fixed in main for v2.3.

- CVE-2020-10063
- Zephyr project bug tracker ZEPSEC-55
- PR24435 fix in main for v2.3
- PR24531 fix for branch from v2.2
- PR24535 fix for branch from v2.1
- PR24530 fix for branch from v1.14
- NCC-ZEP report (NCC-ZEP-032)

CVE-2020-10064

Improper Input Frame Validation in ieee802154 Processing

- CVE-2020-10064
- Zephyr project bug tracker ZEPSEC-65
- PR24971 fix for v2.4
- PR33451 fix for v1.4

CVE-2020-10065

OOB Write after not validating user-supplied length (<= 0xffff) and copying to fixed-size buffer (default: 77 bytes) for HCI_ACL packets in bluetooth HCI over SPI driver.

- CVE-2020-10065
- Zephyr project bug tracker ZEPSEC-66
- This issue has not been fixed.

CVE-2020-10066

Incorrect Error Handling in Bluetooth HCI core

In hci_cmd_done, the buf argument being passed as null causes nullpointer dereference.

- CVE-2020-10066
- Zephyr project bug tracker ZEPSEC-67
- PR24902 fix for v2.4
- PR25089 fix for v1.4

CVE-2020-10067

Integer Overflow In is_in_region Allows User Thread To Access Kernel Memory

A malicious userspace application can cause a integer overflow and bypass security checks performed by system call handlers. The impact would depend on the underlying system call and can range from denial of service to information leak to memory corruption resulting in code execution within the kernel.

See NCC-ZEP-005

This has been fixed in releases v1.14.2, and v2.2.0.

- CVE-2020-10067
- Zephyr project bug tracker ZEPSEC-27
- PR23653 fix for v1.14.2
- PR23654 fix for the v2.1 branch
- PR23239 fix for v2.2.0

CVE-2020-10068

Zephyr Bluetooth DLE duplicate requests vulnerability

In the Zephyr project Bluetooth subsystem, certain duplicate and back-to-back packets can cause incorrect behavior, resulting in a denial of service.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- CVE-2020-10068
- Zephyr project bug tracker ZEPSEC-78
- PR23707 fix for v2.3 (split driver)
- PR23708 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23964 fix for v2.2.0

CVE-2020-10069

Zephyr Bluetooth unchecked packet data results in denial of service

An unchecked parameter in bluetooth data can result in an assertion failure, or division by zero, resulting in a denial of service attack.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- CVE-2020-10069

- Zephyr project bug tracker ZEPSEC-81
- PR23705 fix for v2.3 (split driver)
- PR23706 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23963 fix for branch from v2.2.0

CVE-2020-10070

MQTT buffer overflow on receive buffer

In the Zephyr Project MQTT code, improper bounds checking can result in memory corruption and possibly remote code execution. NCC-ZEP-031

When calculating the packet length, arithmetic overflow can result in accepting a receive buffer larger than the available buffer space, resulting in user data being written beyond this buffer.

This has been fixed in main for v2.3.

- CVE-2020-10070
- Zephyr project bug tracker ZEPSEC-85
- commit 0b39cbf3 for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE-2020-10071

Insufficient publish message length validation in MQTT

The Zephyr MQTT parsing code performs insufficient checking of the length field on publish messages, allowing a buffer overflow and potentially remote code execution. NCC-ZEP-031

This has been fixed in main for v2.3.

- CVE-2020-10071
- Zephyr project bug tracker ZEPSEC-86
- commit 989c4713 fix for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE-2020-10072

All threads can access all socket file descriptors

There is no management of permissions to network socket API file descriptors. Any thread running on the system may read/write a socket file descriptor knowing only the numerical value of the file descriptor.

- CVE-2020-10072
- Zephyr project bug tracker ZEPSEC-87
- PR25804 fix for v2.4
- PR27176 fix for v1.4

CVE-2020-10136

IP-in-IP protocol routes arbitrary traffic by default zephyrproject

- CVE-2020-10136
- Zephyr project bug tracker ZEPSEC-64

CVE-2020-13598

FS: Buffer Overflow when enabling Long File Names in FAT_FS and calling fs_stat

Performing fs_stat on a file with a filename longer than 12 characters long will cause a buffer overflow.

- CVE-2020-13598
- Zephyr project bug tracker ZEPSEC-88
- PR25852 fix for v2.4
- PR28782 fix for v2.3
- PR33577 fix for v1.4

CVE-2020-13599

Security problem with settings and littlefs

When settings is used in combination with littlefs all security related information can be extracted from the device using MCUmgr and this could be used e.g in bt-mesh to get the device key, network key, app keys from the device.

- CVE-2020-13599
- Zephyr project bug tracker ZEPSEC-57
- PR26083 fix for v2.4

CVE-2020-13600

Malformed SPI in response for eswifi can corrupt kernel memory

- CVE-2020-13600
- Zephyr project bug tracker ZEPSEC-91
- PR26712 fix for v2.4

CVE-2020-13601

Possible read out of bounds in dns read

- CVE-2020-13601
- Zephyr project bug tracker ZEPSEC-92
- PR27774 fix for v2.4
- PR30503 fix for v1.4

CVE-2020-13602

Remote Denial of Service in LwM2M do_write_op_tlv

In the Zephyr LwM2M implementation, malformed input can result in an infinite loop, resulting in a denial of service attack.

- CVE-2020-13602
- Zephyr project bug tracker ZEPSEC-56
- PR26571 fix for v2.4
- PR33578 fix for v1.4

CVE-2020-13603

Possible overflow in mempool

- Zephyr offers pre-built 'malloc' wrapper function instead.
- The 'malloc' function is wrapper for the 'sys_mem_pool_alloc' function
- sys_mem_pool_alloc allocates 'size + WB_UP(sizeof(struct sys_mem_pool_block))' in an unsafe manner.
- Asking for very large size values leads to internal integer wrap-around.
- Integer wrap-around leads to successful allocation of very small memory.
- For example: calling malloc(0xffffffff) leads to successful allocation of 7 bytes.
- That leads to heap overflow.
- CVE-2020-13603
- Zephyr project bug tracker ZEPSEC-111
- PR31796 fix for v2.4
- PR32808 fix for v1.4

9.6.4 CVE-2021

CVE-2021-3319

DOS: Incorrect 802154 Frame Validation for Omitted Source / Dest Addresses

Improper processing of omitted source and destination addresses in ieee802154 frame validation (ieee802154_validate_frame)

This has been fixed in main for v2.5.0

- CVE-2020-3319
- Zephyr project bug tracker GHSA-94jg-2p6q-5364
- PR31908 fix for main

CVE-2021-3320

Mismatch between validation and handling of 802154 ACK frames, where ACK frames are considered during validation, but not during actual processing, leading to a type confusion.

- CVE-2020-3320
- PR31908 fix for main

CVE-2021-3321

Incomplete check of minimum IEEE 802154 fragment size leading to an integer underflow.

- CVE-2020-3321
- Zephyr project bug tracker ZEPSEC-114
- PR33453 fix for v2.4

CVE-2021-3323

Integer Underflow in 6LoWPAN IPHC Header Uncompression

- CVE-2020-3323
- Zephyr project bug tracker ZEPSEC-116
- This issue has not been fixed.

CVE-2021-3430

Assertion reachable with repeated LL_CONNECTION_PARAM_REQ.

This has been fixed in main for v2.6.0

- CVE-2021-3430
- Zephyr project bug tracker GHSA-46h3-hjcq-2jjr
- PR 33272 fix for main
- PR 33369 fix for 2.5
- PR 33759 fix for 1.14.2

CVE-2021-3431

BT: Assertion failure on repeated LL_FEATURE_REQ

This has been fixed in main for v2.6.0

- CVE-2021-3431
- Zephyr project bug tracker GHSA-7548-5m6f-mqv9
- PR 33340 fix for main
- PR 33369 fix for 2.5

CVE-2021-3432

Invalid interval in CONNECT_IND leads to Division by Zero

This has been fixed in main for v2.6.0

- CVE-2021-3432
- Zephyr project bug tracker GHSA-7364-p4wc-8mj4
- PR 33278 fix for main
- PR 33369 fix for 2.5

CVE-2021-3433

BT: Invalid channel map in CONNECT_IND results to Deadlock

This has been fixed in main for v2.6.0

- CVE-2021-3433
- Zephyr project bug tracker GHSA-3c2f-w4v6-qxrp
- PR 33278 fix for main
- PR 33369 fix for 2.5

CVE-2021-3434

L2CAP: Stack based buffer overflow in le_ecred_conn_req()

This has been fixed in main for v2.6.0

- CVE-2021-3434
- Zephyr project bug tracker GHSA-8w87-6rfp-cfrm
- PR 33305 fix for main
- PR 33419 fix for 2.5
- PR 33418 fix for 1.14.2

CVE-2021-3435

L2CAP: Information leakage in le_ecred_conn_req()

This has been fixed in main for v2.6.0

- CVE-2021-3435
- Zephyr project bug tracker GHSA-xhg3-gvj6-4rqh
- PR 33305 fix for main
- PR 33419 fix for 2.5
- PR 33418 fix for 1.14.2

CVE-2021-3436

Bluetooth: Possible to overwrite an existing bond during keys distribution phase when the identity address of the bond is known

During the distribution of the identity address information we don't check for an existing bond with the same identity address.This means that a duplicate entry will be created in RAM while the newest entry will overwrite the existing one in persistent storage.

This has been fixed in main for v2.6.0

- CVE-2021-3436
- Zephyr project bug tracker GHSA-j76f-35mc-4h63
- PR 33266 fix for main
- PR 33432 fix for 2.5
- PR 33433 fix for 2.4
- PR 33718 fix for 1.14.2

CVE-2021-3454

Truncated L2CAP K-frame causes assertion failure

For example, sending L2CAP K-frame where SDU length field is truncated to only one byte, causes assertion failure in previous releases of Zephyr. This has been fixed in master by commit 0ba9437 but has not yet been backported to older release branches.

This has been fixed in main for v2.6.0

- CVE-2021-3454
- Zephyr project bug tracker GHSA-fx88-6c29-vrp3
- PR 32588 fix for main
- PR 33513 fix for 2.5
- PR 33514 fix for 2.4

CVE-2021-3455

Disconnecting L2CAP channel right after invalid ATT request leads freeze

When Central device connects to peripheral and creates L2CAP connection for Enhanced ATT, sending some invalid ATT request and disconnecting immediately causes freeze.

This has been fixed in main for v2.6.0

- CVE-2021-3455
- Zephyr project bug tracker GHSA-7g38-3x9v-v7vp
- PR 35597 fix for main
- PR 36104 fix for 2.5
- PR 36105 fix for 2.4

CVE-2021-3510

Zephyr JSON decoder incorrectly decodes array of array

When using JSON_OBJ_DESCR_ARRAY_ARRAY, the subarray is has the token type JSON_TOK_LIST_START, but then assigns to the object part of the union. arr_parse then takes the offset of the array-object (which has nothing todo with the list) treats it as relative to the parent object, and stores the length of the subarray in there.

This has been fixed in main for v2.7.0

- CVE-2021-3510
- Zephyr project bug tracker GHSA-289f-7mw3-2qf4
- PR 36340 fix for main
- PR 37816 fix for 2.6

CVE-2021-3581

HCI data not properly checked leads to memory overflow in the Bluetooth stack

In the process of setting SCAN_RSP through the HCI command, the Zephyr Bluetooth protocol stack did not effectively check the length of the incoming HCI data. Causes memory overflow, and then the data in the memory is overwritten, and may even cause arbitrary code execution.

This has been fixed in main for v2.6.0

- CVE-2021-3581
- Zephyr project bug tracker GHSA-8q65-5gqf-fmw5
- PR 35935 fix for main
- PR 35984 fix for 2.5
- PR 35985 fix for 2.4
- PR 35985 fix for 1.14

CVE-2021-3625

Buffer overflow in Zephyr USB DFU DNLOAD

This has been fixed in main for v2.6.0

- CVE-2021-3625
- Zephyr project bug tracker GHSA-c3gr-hgvr-f363
- PR 36694 fix for main

Bibliography

[th-imboot] Must boot with an immutable bootloader.

[th-authrepl] Application image shall only be replaced with an authorized image.

[th-timely-update] Application updates shall be done in a timely manner.

[th-atomic-update] Application updates shall be atomic.

[th-root-certs] TLS must have a list of trusted root certificates.

[th-root-check] TLS must verify root certificate from server is valid.

[th-secret-storage] There must be a mechanism to securely store client secrets. The least amount of code necessary shall have access to these secrets.

[th-time] System must have moderately accurate notion of the current date/time.

[th-conf] The system must receive, and keep configuration data.

[th-logs] The system must log security-related events, and either store them locally, or send to a service.

[th-all-tls] All communications with the cloud service shall use TLS.

[th-tls-ciphers] TLS shall be configured to allow only generally agreed cipher suites (including forward secrecy).

[th-tls-client-auth] The device shall authenticate itself with the cloud provider using one of the methods described.

[th-entropy] The TLS layer shall use a modern, accepted cryptographic random-bit generator seeded by an entropy source within the SoC.

[th-initial-provision] The device shall have a per-device secret loaded before deployment.

[th-initial-secret] The initial secret shall be securely maintained, and destroyed in any external location as soon as the device is provisioned.

[th-reprovision] Reprovisioning a device shall be done securely.

[th-destruction] Upon decommissioning, the device secret shall be rendered ineffective.

Python Module Index

r

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z