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:

- ARCv2 (EM and HS) and ARCv3 (HS6X)
- 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)
- MIPS (MIPS32 Release 1 specification)
- 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 overarching, application or policy-defined, System Power Management and fine-grained, driver-defined, Device Power Management.

**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
- Generic Attribute Profile (GATT)
- 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 port (native_sim) lets you build and run Zephyr as a native application on Linux, aiding development and testing.

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

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 port
Native sim allows 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:
1.4.1  Getting Started

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

1.4.2  Code and Development

- Source Code Repository
- Releases
- Contribution Guide

1.4.3  Community and Support

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

1.4.4  Issue Tracking and Security

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

1.4.5  Additional Resources

- Zephyr Project Website
- Zephyr Tech Talks

1.5  Fundamental Terms and Concepts

See Glossary of Terms
Chapter 2

Developing with Zephyr

2.1 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

2.1.1 Select and Update OS

Click the operating system you are using.

Ubuntu

This guide covers Ubuntu version 20.04 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.

2.1.2 Install dependencies

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

The current minimum required version for the main dependencies are:
<table>
<thead>
<tr>
<th>Tool</th>
<th>Min. Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMake</td>
<td>3.20.5</td>
</tr>
<tr>
<td>Python</td>
<td>3.8</td>
</tr>
<tr>
<td>Devicetree compiler</td>
<td>1.4.6</td>
</tr>
</tbody>
</table>

Ubuntu

1. If using an Ubuntu version older than 22.04, it is necessary to add extra repositories to meet the minimum required versions for the main dependencies listed above. In that case, 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:

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

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

   ```bash
   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 libssl12-dev libmagic1
   ```

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

   ```bash
   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:

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

2. After the Homebrew installation script completes, follow the on-screen instructions to add the Homebrew installation to the path.
   - On macOS running on Apple Silicon, this is achieved with:
     ```bash
     (echo; echo 'export PATH=\"$(/opt/homebrew/bin/brew shellenv)\"\"') >> ~/.zprofile
     source ~/.zprofile
     ```
   - On macOS running on Intel, use the command for Apple Silicon, but replace `/opt/homebrew` with `/usr/local`.

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

   ```bash
   brew install cmake ninja gperf python3 ccache qemu dtc libmagic1 wget
   ```

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

   ```bash
   (echo; echo 'export PATH="$(brew --prefix)/opt/python/libexec/bin:$PATH"')
   ...>> ~/.zprofile
   source ~/.zprofile
   ```
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 terminal window. In modern version of Windows (10 and later) it is recommended to install the Windows Terminal application from the Microsoft Store. 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` terminal window as **Administrator**. To do so, press the Windows key, type `cmd.exe`, right-click the Command Prompt search 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 python311 git dtc-msys2 wget 7zip
   ```

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

5. Close the terminal window.

**2.1.3 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.

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

For this reason it is suggested to use Python virtual environments.

Ubuntu

Install within virtual environment

1. Use `apt` to install Python venv package:
   ```
   sudo apt install python3-venv
   ```
2. Create a new virtual environment:
3. Activate the virtual environment:

```bash
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.

4. Install west:

```bash
pip install west
```

5. Get the Zephyr source code:

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

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

```bash
west zephyr-export
```

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

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

### Install globally

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

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

2. Get the Zephyr source code:

```bash
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.

```bash
west zephyr-export
```

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

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

### macOS

**Install within virtual environment**

1. Create a new virtual environment:
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:

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

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

```
w 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
```

---

### Install globally

1. Install west:

```
pip3 install -U west
```

2. Get the Zephyr source code:

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

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

```
w 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
```

---

### Windows

Install within virtual environment

1. Open a `cmd.exe` terminal window **as a regular user**

2. Create a new virtual environment:
3. Activate the virtual environment:

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

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.

4. Install west:

```
pip install west
```

5. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

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

**Install globally**

1. Open a `cmd.exe` terminal window **as a regular user**

2. Install west:

```
pip3 install -U west
```

3. Get the Zephyr source code:

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

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

```
west zephyr-export
```

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

```
pip3 install -r %HOMEPATH%\zephyrproject\zephyr\scripts\requirements.txt
```
2.1.4 Install the Zephyr SDK

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

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

Note: You can change 0.16.5-1 to another version in the instructions below if needed; the Zephyr SDK Releases page contains all available SDK releases.

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

Ubuntu

1. Download and verify the Zephyr SDK bundle:
   ```
   cd ~
   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/
   ...zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
   wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.
   ...16.5-1/sha256.sum | shasum --check --ignore-missing
   ```
   If your host architecture is 64-bit ARM (for example, Raspberry Pi), replace x86_64 with aarch64 in order to download the 64-bit ARM Linux SDK.

2. Extract the Zephyr SDK bundle archive:
   ```
   tar xvf zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
   ```
   Note: It is recommended to extract the Zephyr SDK bundle at one of the following locations:
   - $HOME
   - $HOME/.local
   - $HOME/.local/opt
   - $HOME/bin
   - /opt
   - /usr/local
   The Zephyr SDK bundle archive contains the zephyr-sdk-<version> directory and, when extracted under $HOME, the resulting installation path will be $HOME/zephyr-sdk-<version>.

3. Run the Zephyr SDK bundle setup script:
   ```
   cd zephyr-sdk-0.16.5-1
   ./setup.sh
   ```
   Note: You only need to run the setup script once after extracting the Zephyr SDK bundle.
   You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

4. Install udev rules, which allow you to flash most Zephyr boards as a regular user:
sudo cp ~/zephyr-sdk-0.16.5-1/sysroots/x86_64-pokysdk-linux/usr/share/openocd/...contrib/60-openocd.rules /etc/udev/rules.d
sudo udevadm control --reload

macOS

1. Download and verify the Zephyr SDK bundle:
   cd ~
curl -L -O https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/zephyr-sdk-0.16.5-1_macos-x86_64.tar.xz
curl -L https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/sha256.sum | shasum --check --ignore-missing

   If your host architecture is 64-bit ARM (Apple Silicon, also known as M1), replace x86_64 with aarch64 in order to download the 64-bit ARM macOS SDK.

2. Extract the Zephyr SDK bundle archive:
   tar xvf zephyr-sdk-0.16.5-1_macos-x86_64.tar.xz

   **Note:** It is recommended to extract the Zephyr SDK bundle at one of the following locations:
   - $HOME
   - $HOME/.local
   - $HOME/.local/opt
   - $HOME/bin
   - /opt
   - /usr/local

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

3. Run the Zephyr SDK bundle setup script:
   cd zephyr-sdk-0.16.5-1
   ./setup.sh

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

Windows

1. Open a cmd.exe terminal window as a regular user
2. Download the Zephyr SDK bundle:
   cd %HOMEPATH%
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/zephyr-sdk-0.16.5-1_windows-x86_64.7z
3. Extract the Zephyr SDK bundle archive:
   7z x zephyr-sdk-0.16.5-1_windows-x86_64.7z

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

4. Run the Zephyr SDK bundle setup script:
   ```
   cd zephyr-sdk-0.16.5-1
   setup.cmd
   ```

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

### 2.1.5 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.

If you are unsure what name west uses for your board, west boards can be used to obtain a list of all boards Zephyr supports.

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

**Ubuntu**

```

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

**macOS**

```

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

**Windows**

```

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

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

**Note:** A board may contain one or multiple SoCs, Also, each SoC may contain one or more CPU clusters. When building for such boards it is necessary to specify the SoC or CPU cluster for which the sample must be built. For example to build blinky for the cpuapp core on the nRF5340DK the board must be provided as: `nrf5340dk/nrf5340/cpuapp`. See also **Board terminology** for more details.
2.1.6 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:

![Phytec reel_board running blinky](image)

Fig. 1: Phytec reel_board running blinky

2.1.7 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

2.1.8 Troubleshooting Installation

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

Double Check the Zephyr SDK Variables When Updating

When updating Zephyr SDK, check whether the `ZEPHYR_TOOLCHAIN_VARIANT` or `ZEPHYR_SDK_INSTALL_DIR` environment variables are already set. See Updating the Zephyr SDK toolchain for more information.
For more information about these environment variables in Zephyr, see **Important Environment Variables**.

### 2.1.9 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.

#### 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.)

#### 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. Unnecessary screenshots exclude vision impaired developers; some are major Zephyr contributors. Accessibility has been recognized as a basic human right by the United Nations.

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

### 2.2 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.

#### 2.2.1 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. Pip is Python's package installer. Its `install` command first tries to reuse 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.
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, 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.)

### 2.2.2 Advanced Platform Setup

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

#### 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**

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

**Fedora**

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.
Zephyr Project Documentation, Release 3.6.99

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 \ make gcc gcc-multilib g++-multilib libssl2-dev libmagic1

Fedora

sudo dnf group install "Development Tools" "C Development Tools and Libraries" sudo dnf install cmake ninja-build gperf dfu-util dtc wget which \ python3-pip python3-tkinter xz file python3-devel 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 un-install the previous version of cmake.)

2.2. Beyond the Getting Started Guide 17
cd ~
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 $HOME/bin/cmake
yes | sh cmake-3.21.1-Linux-x86_64.sh | cat
echo "export PATH=$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)  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 and OpenOCD.

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

The Zephyr SDK supports the following target architectures:

• ARC (32-bit and 64-bit; AR Cv1, AR Cv2, AR Cv3)
• ARM (32-bit and 64-bit; ARMv6, ARMv7, ARMv8; A/R/M Profiles)
Follow these steps to install the Zephyr SDK:

1. Download and verify the Zephyr SDK bundle:
   
   ```
   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
   wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/sha256.sum | shasum --check --ignore-missing
   ```

   You can change 0.16.5-1 to another version if needed; the Zephyr SDK Releases page contains all available SDK releases.

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

2. Extract the Zephyr SDK bundle archive:

   ```
   cd <sdk download directory>
   tar xvf zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
   ```

3. Run the Zephyr SDK bundle setup script:

   ```
   cd zephyr-sdk-0.16.5-1
   ./setup.sh
   ```

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

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

---

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

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

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

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

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

- `/company/tools/zephyr-sdk-0.13.2`
- `/company/tools/zephyr-sdk-a.b.c`
- `/company/tools/zephyr-sdk-x.y.z`
This allows the Zephyr build system to choose the correct version of the SDK, while allowing multiple Zephyr SDKs to be grouped together at a specific path.

### 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 prebuilt host tools. It is, however, possible to build without the SDK's toolchain by using another toolchain as described in the **Toolchains** section.

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:

```bash
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 **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](https://github.com/zephyrproject/zephyr/issues/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.

### 2.2.3 Install a Toolchain

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

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

You can 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.

### 2.2.4 Updating the Zephyr SDK toolchain

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

- **If the variables are not set,** the latest compatible version of Zephyr SDK will be selected by default. Proceed to next step without making any changes.

- **If `ZEPHYR_TOOLCHAIN_VARIANT` is set,** the corresponding toolchain will be selected at build time. Zephyr SDK is identified by the value `zephyr`. If the `ZEPHYR_TOOLCHAIN_VARIANT` environment variable is not `zephyr`, then either unset it or change its value to `zephyr` to make sure Zephyr SDK is selected.

- **If the `ZEPHYR_SDK_INSTALL_DIR` environment variable is set,** it will override the default lookup location for Zephyr SDK. If you install Zephyr SDK to one of the recommended locations, you can unset this variable. Otherwise, set it to your chosen install location.

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

### 2.2.5 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.

```
w 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
```

### 2.2.6 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*.

### 2.2.7 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/nrf51822)
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`.

### 2.2.8 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 application with `native_sim`. Additional information about building applications can be found in the *Building an Application* section.

**Build Blinky**

Let's build the blinky sample application.
Zephyr applications are built to run on specific hardware, called a “board”. 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 reuse 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.

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:

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.

2.2. Beyond the Getting Started Guide 23
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.)

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

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 (Linux)

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

First, build Hello World for native_sim.

```
# From the root of the zephyr repository
west build -b native_sim 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.
2.3 Environment Variables

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

2.3.1 Setting Variables

Option 1: Just Once

To set the environment variable \texttt{MY\_VARIABLE} to \texttt{foo} for the lifetime of your current terminal window:

\textbf{Linux/macOS}

\begin{verbatim}
export MY_VARIABLE=foo
\end{verbatim}

\textbf{Windows}

\begin{verbatim}
set MY_VARIABLE=foo
\end{verbatim}

\textbf{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.

Option 2: In all Terminals

\textbf{Linux/macOS}

Add the \texttt{export MY\_VARIABLE=foo} line to your shell's startup script in your home directory. For Bash, this is usually \texttt{~/.bashrc} on Linux or \texttt{~/.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.

\textbf{Windows}

You can use the \texttt{setx} program in \texttt{cmd.exe} or the third-party RapidEE program.

To use \texttt{setx}, type this command, then close the terminal window. Any new \texttt{cmd.exe} windows will have \texttt{MY\_VARIABLE} set to \texttt{foo}.

\begin{verbatim}
setx MY_VARIABLE foo
\end{verbatim}

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

\begin{verbatim}
choco install rapidee
\end{verbatim}

You can then run \texttt{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.
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.

Linux/macOS

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` 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.

2.3.2 Zephyr Environment Scripts

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

Linux/macOS

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

Windows

```
zephyr-env.cmd
```

These scripts:

- set `ZEPHYR_BASE` 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.
2.3.3 Important Environment Variables

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

**BOARD**
- See Important Build System Variables.

**CONF_FILE**
- See Important Build System Variables.

**SHIELD**
- See Shields.

**ZEPHYR_BASE**
- See Important Build System Variables.

**EXTRA_ZEPHYR_MODULES**
- See Important Build System Variables.

**ZEPHYR_MODULES**
- See Important Build System Variables.

**ZEPHYR_BOARD_ALIASES**
- See Board Aliases.

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

**ZEPHYR_SDK_INSTALL_DIR**
- Path where Zephyr SDK is installed.

**ZEPHYR_TOOLCHAIN_VARIANT**
The name of the toolchain to use.

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

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

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

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

## 2.4 Application Development

**Note:** In this document, we'll assume:

- your application directory, `<app>`, is something like `<home>/zephyrproject/app`
- its build directory is `<app>/build`
These terms are defined below. On Linux/macOS, `<home>` is equivalent to `~`. On Windows, it's `%userprofile%`.

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

### 2.4.1 Overview

Zephyr's build system is based on CMake.

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

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

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

Here are the files in a simple Zephyr application:

```
<app>
  ├── CMakeLists.txt
  │   └── app.overlay
  │       └── prj.conf
  │       ├── VERSION
  │       └── 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 configuration files, the ability to run and debug compiled binaries on real or emulated hardware, and more.

- **app.overlay**: This is a devicetree overlay file that specifies application-specific changes which should be applied to the base devicetree for any board you build for. The purpose of devicetree overlays is usually to configure something about the hardware used by the application.

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

  See **Devicetree** for more information about devicetree.

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

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

  See **Kconfig Configuration** below for more information.

- **VERSION**: A text file that contains several version information fields. These fields let you manage the lifecycle of the application and automate providing the application version when signing application images.
See *Application version management* for more information about this file and how to use it.

- **main.c**: A source code file. Applications typically contain source files written in C, C++, or assembly language. The Zephyr convention is to place them in a subdirectory of `<app>` named `src`.

Once an application has been defined, you will use CMake to generate a build directory, which contains the files you need to build the application and Zephyr; then link them together into a final binary you can run on your board. The easiest way to do this is with west build, but you can use CMake directly also. Application build artifacts are always generated in a separate 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.

### 2.4.2 Application types

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

<table>
<thead>
<tr>
<th>Application type</th>
<th><code>&lt;app&gt;</code> location</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository</td>
<td>zephyr repository</td>
</tr>
<tr>
<td>workspace</td>
<td>west workspace where Zephyr is installed</td>
</tr>
<tr>
<td>freestanding</td>
<td>other locations</td>
</tr>
</tbody>
</table>

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

#### Zephyr repository application

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

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

#### Zephyr workspace application

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

```
zephyrproject/
    └─── .west/
            └─── config
        └─── zephyr/
```

(continues on next page)
Zephyr freestanding application

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

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

2.4.3 Creating an Application

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

Using a Reference Workspace Application

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

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

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

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

```
cd <home>/zephyrproject
git clone https://github.com/zephyrproject-rtos/example-application my-app
```
The directory name `my-app` above is arbitrary: change it as needed. You can now go into this directory and adapt its contents to suit your needs. Since you are using an existing Zephyr workspace, you can use `west build` or any other west commands to build, flash, and debug.

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

- remove Zephyr modules you don't need
- add additional custom repositories of your own
- override repositories provided by Zephyr with your own versions
- share the results with others and collaborate further

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

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

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

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

When you are satisfied, you can run:

```
west update
```

and your workspace will be ready for use.

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

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

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

**Creating an Application by Hand**

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

1. Create an application directory.

   For example, in a Unix shell or Windows `cmd.exe` prompt:
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. Create your source code files.

   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:

   ```cmake
   cmake_minimum_required(VERSION 3.20.0)
   find_package(Zephyr)
   project(my_zephyr_app)
   target_sources(app PRIVATE src/main.c)
   ```

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

5. Create at least one Kconfig fragment for your application (usually named `prj.conf`) and set Kconfig option values needed by your application there. See Kconfig Configuration. If no Kconfig options need to be set, create an empty file.

6. Configure any devicetree overlays needed by your application, usually in a file named `app.overlay`. See Set devicetree overlays.

7. Set up any other files you may need, such as `twister` configuration files, continuous integration files, documentation, etc.

### 2.4.4 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.
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 **Environment 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.
- **EXTRA_CONF_FILE**: 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 device-tree** for information about devicetree and Zephyr.
- **EXTRA_DTC_OVERLAY_FILE**: Additional devicetree overlay files to use. Multiple files can be separated with semicolons. This can be useful to leave **DTC_OVERLAY_FILE** at its default value, but “mix in” some additional overlay files.
- **SHIELD**: see **Shields**
- **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. If you set this variable, it must be a complete list of all modules to use, as the build system will not automatically pick up any modules from west.
- **EXTRA_ZEPHYR_MODULES**: Like **ZEPHYR_MODULES**, except these will be added to the list of modules found via west, instead of replacing it.
- **FILE_SUFFIX**: Optional suffix for filenames that will be added to Kconfig fragments and devicetree overlays (if these files exists, otherwise will fallback to the name without the prefix). See **File Suffixes** for details.

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

### 2.4.5 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:

```cmake
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:

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

Below is a simple example CMakeList.txt:

```cmake
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:

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

### 2.4.6 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 running cmake.

### 2.4.7 Application Configuration

2.4. Application Development
Application Configuration Directory

Zephyr will use configuration files from the application's configuration directory except for files with an absolute path provided by the arguments described earlier, for example CONF_FILE, EXTRA_CONF_FILE, DTC_OVERLAY_FILE, and EXTRA_DTC_OVERLAY_FILE. The application configuration directory is defined by the APPLICATION_CONFIG_DIR variable. APPLICATION_CONFIG_DIR will be set by one of the sources below with the highest priority listed first.

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

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:

```bash
CONFIG_CPP=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 kconfig-search 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.

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

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

```bash
CONFIG_WARN_EXPERIMENTAL=y
```

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

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

Devicetree Overlays

See Set devicetree overlays.

File Suffixes

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

Given the following example project layout:

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

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

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

### 2.4.8 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.

#### 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/modules/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.

### 2.4.9 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:

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

Using CMake and ninja:

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

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

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:

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

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

### Basics

1. Navigate to the application directory `<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:
Using CMake and ninja:

```bash
mkdir build && cd build

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

# Now run the build tool 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:

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

Using CMake and ninja:

```bash
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**.

### Build Directory Contents

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

```
<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.

### 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 `<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.

   ```shell
   west build -t clean
   ```

   or

   ```shell
   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.

   ```shell
   west build -t pristine
   ```

   or

   ```shell
   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.

### 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`: 
Using CMake and ninja:

```bash
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
```

### 2.4.10 Run an Application

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

**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, `<app>/build`, to flash the compiled Zephyr image and run it on your board:

   ```bash
   west flash
   ```

   or

   ```bash
   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.

**Running in an Emulator**

Zephyr has built-in emulator support for QEMU. It allows you to run and test an application virtually, before (or in lieu of) loading and running it on actual target hardware.
Check out *Beyond the Getting Started Guide* for additional steps needed on Windows.

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, `<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.

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

### 2.4.11 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
```

(continues on next page)
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.

**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/
- 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:

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

Using CMake and ninja:

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

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.

**SOC Definitions**

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

![Diagram of the file structure]

The Kconfig files above may describe the SoC or load additional SoC Kconfig files. An example of loading `stm31l0` specific Kconfig files in this structure:

```plaintext
soc
  └── arm
      └── st_stm32
          ├── Kconfig
          │     ├── Kconfig.soc
          │     └── Kconfig.defconfig
          └── stm31l0
              └── Kconfig.series
```

can be done with the following content in `st_stm32/Kconfig.soc`:

```plaintext
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:

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

Using CMake and ninja:

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

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

2.5 Debugging

2.5.1 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 ELF (Executable and Linkable Format) 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 `CONFIG_KERNEL_BIN_NAME`.

**GDB server**

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. There are multiple ways to do this. Each way starts a QEMU instance with the processor halted at startup and with a GDB server instance listening for a connection.

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

```bash
dev kingsize: locals $QEMU_PIPE
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.

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

```bash
ninja debugserver
```

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

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

```bash
west build -t debugserver_qemu
```

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

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

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

**GDB client**

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

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

**Note:** Substitute the correct `ZEPHYR_BASE` for your system.

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

**Note:** The GDB version on your 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.

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

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

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

`ddd` 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.

**Debugging**

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

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

### 2.5.2 Debug with Eclipse

**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.

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.

Generate and Import an Eclipse Project

1. Set up a GNU Arm Embedded toolchain as described in GNU Arm Embedded.
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 -Bbuild -GNinja -DBOARD=frdm_k64f -G"Eclipse CDT4 - Ninja" %ZEPHYR_BASE%\samples\synchronization
   ninja -Cbuild
   ```

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.

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.

**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.

**2.5.3 Debugging I2C communication**

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

The sample output of the dump looks like this:

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

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

  **Note:** the address, length and contents values are in hexadecimal, but lack the 0x prefix.
Next lines contain messages, both sent and received. The contents of write messages is always shown, while the content of read messages is controlled by a parameter to the function \texttt{i2c\_dump\_msgs\_rw}. This function is available for use by user, but is also called internally by \texttt{i2c\_transfer} API function with read content dump enabled. Before the length parameter, the header of the message is printed using abbreviations:

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

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

### Filtering the I2C communication dump

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

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

```c
/ {
  i2c {
    display0: some-display@a {
      ...
    };
    sensor3: some-sensor@b {
      ...
    };
  };
  i2c-dump-allowlist {
    compatible = "zephyr,i2c-dump-allowlist";
    devices = < &display0 >, < &sensor3 >;
  };
};
```

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

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

## 2.6 API Status and Guidelines

### 2.6.1 API Overview

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

The version column uses \texttt{semantic version}, and has the following expectations:
• Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.
  – If minor version is up to one (0.1.z), API is considered experimental.
  – If minor version is larger than one (0.y.z | y > 1), API is considered unstable.

• Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.
  – APIs with major versions equal or larger than one (x.y.z | x >= 1) are considered stable.
  – All existing stable APIs in Zephyr will be start with version 1.0.0.

• Patch version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.

• Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.

• Major version X (x.Y.z | x > 0) MUST be incremented if a compatibility breaking change was made to the API.

Note: Version for existing APIs are initially set based on the current state of the APIs:
  • 0.1.0 denotes an experimental API
  • 0.8.0 denote an unstable API,
  • and finally 1.0.0 indicates a stable APIs.

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

<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio Codec Interface</td>
<td>0.1.0</td>
<td>v1.13.0</td>
</tr>
<tr>
<td>Digital Microphone Interface</td>
<td>0.1.0</td>
<td>v1.13.0</td>
</tr>
<tr>
<td>Bindesc Define Connectivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUDIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute Protocol (ATT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Audio Input Control Service (AICS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Battery Service (BAS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Audio</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codec capability parsing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codec config parsing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Basic Audio Profile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAP Broadcast APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAP Broadcast Sink APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAP Broadcast Source APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAP Broadcast Sink APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAP Broadcast Source APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAP Unicast Client APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAP Unicast Server APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Gaming Audio Profile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh BLOB Transfer Client model API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh BLOB Transfer Server model API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh BLOB flash stream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh BLOB model API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh Device Firmware Update</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh Device Firmware Update (DFU) metadata</td>
<td>1.0.0</td>
<td></td>
</tr>
<tr>
<td>Firmware Update Server model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh On-Demand Private GATT Proxy Client</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh On-Demand Private GATT Proxy Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh Private Beacon Client</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh Private Beacon Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh SAR Configuration Client Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh SAR Configuration Server Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh Solicitation PDU RPL Client</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth Mesh Solicitation PDU RPL Server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration Client Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration Server Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Firmware Distribution models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Firmware Distribution Server model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health Client Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health Server Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health faults</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heartbeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Large Composition Data Client model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Large Composition Data Server model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcodes Aggregator Client model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcodes Aggregator Server model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provisioning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Provisioning Client model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Provisioning models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote provisioning server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Runtime Configuration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Configuration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_subnet Configuration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR Configuration common header</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statistic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth testing callbacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byteorder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common Audio Profile (CAP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coordinated Set Identification Profile (CSIP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cryptography</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data buffers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encrypted Advertising Data (EAD)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic Access Profile (GAP)</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Defines and Assigned Numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic Attribute Profile (GATT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GATT Client APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GATT Server APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HCI RAW channel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCI drivers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hands Free Profile (HFP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hearing Access Service (HAS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heart Rate Service (HRS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediate Alert Service (IAS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Isochronous channels (ISO)</td>
<td>0.8.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>L2CAP</td>
<td>0.8.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Media Control Client (MCC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Media Control Service (MCS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Media Proxy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microphone Control Profile (MICP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Transfer Service (OTS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public Broadcast Profile (PBP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RFCOMM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service Discovery Protocol (SDP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UUIDs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Volume Control Profile (VCP)</td>
<td>0.1.0</td>
<td>v2.5.0</td>
</tr>
<tr>
<td>Volume Offset Control Service (VOCS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN ISO-TP Protocol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEEE 802.15.4 and Thread APIs</td>
<td>0.8.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>IEEE 802.15.4 Drivers</td>
<td>0.8.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>IEEE 802.15.4 L2</td>
<td>0.8.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>IEEE 802.15.4 Net Management</td>
<td>0.8.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>OpenThread L2 abstraction layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoRaWAN APIs</td>
<td>0.1.0</td>
<td>v2.5.0</td>
</tr>
<tr>
<td>Networking</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Application network context</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSD Sockets compatible API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket options for TLS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSD socket service API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COAP Library</td>
<td>0.8.0</td>
<td>v1.10.0</td>
</tr>
<tr>
<td>CoAP Manager Events</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoAP client API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoAP service API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection Manager API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection Manager Connectivity API</td>
<td>0.1.0</td>
<td>v3.4.0</td>
</tr>
<tr>
<td>Connection Manager Connectivity Bulk API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection Manager Connectivity Implementation API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DHCPv4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DHCPv4 server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DHCPv6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DNS Resolve Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DNS Service Discovery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed Switch Architecture definitions and helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dummy L2/driver Support Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ethernet Bridging API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ethernet Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ethernet PHY Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ethernet Support Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ethernet MII Support Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEEE 802.3 management interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP client API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP request methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP response status codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IGMP API</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**2.6. API Status and Guidelines**
<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4/IPv6 primitives and helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link Layer Discovery Protocol definitions and helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LwM2M high-level API</td>
<td>0.8.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>LwM2M path helper macros</td>
<td></td>
<td>v1.9.0</td>
</tr>
<tr>
<td>MQTT Client library</td>
<td>0.8.0</td>
<td>v1.14.0</td>
</tr>
<tr>
<td>MQTT-SN Client library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Buffer Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Configuration Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Core Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Core Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Core Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Hostname Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Interface abstraction layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network L2 Abstraction Layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Link Address Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Offloading Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Packet Filter API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Filter Conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ethernet Filter Conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Packet Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Statistics Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network long timeout primitives and helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network packet capture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network time representation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offloaded Net Devices</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPP L2/driver Support Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTP time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Promiscuous mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNTP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send and receive IPv4 or IPv6 ICMP Echo Request messages.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFTP Client library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLS credentials management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trickle Algorithm Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Interface Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual LAN definitions and helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Network Interface Support Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Websocket API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wi-Fi Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wi-Fi Network Manager API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zperf API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gPTP support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gPTP support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB BOS support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB HID class API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HID class USB specific definitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB HID common definitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mouse and keyboard report descriptors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB HID Item helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Host Core API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB device core API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB device core API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP Interface</td>
<td>0.1.0</td>
<td>v3.3.0</td>
</tr>
<tr>
<td>Basic Math Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Absolute Value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Clipping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Dot Product</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Multiplication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Negate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Subtraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector bitwise AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector bitwise NOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector bitwise OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector bitwise XOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Helper macros for printing Q values.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Device Driver APIs

1-Wire Interface

- 1-Wire Sensor API
- 1-Wire data link layer
- 1-Wire network layer

ADC driver APIs

- Emulated ADC

Analog axis API

BBRAM Interface

BBRAM emulator backend API

BC1.2 backed emulator APIs

BC1.2 driver APIs

CAN Interface

- CAN Transceiver

Cache Controller Interface

Cellular Interface

Charger Interface

Clock Control Interface

- LiteX Clock Control driver interface

Coredump pseudo-device driver APIs

Counter Interface

- DAC driver APIs
- DAI Interface
- DMA Interface
- Disk Driver Interface
- Display Interface
- LCD Interface
- EC Host Command Interface
- EDAC API
- EEPROM Interface
- ESPi Driver APIs
- Entropy Interface
- External Cache Controller Interface
- FLASH Interface
- FLASH internal Interface
- Fuel Gauge Interface
- Fuel gauge backend emulator APIs
- GNSS Interface
- GPIO Driver APIs
- Emulated GPIO

nPM1300-specific GPIO Flags

continues on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>nPM6001-specific GPIO Flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nRF-specific GPIO Flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HW spinlock Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Info Interface</td>
<td>1.0.0</td>
<td>v1.14.0</td>
</tr>
<tr>
<td>I2C EEPROM Target Driver API</td>
<td>1.0.0</td>
<td>v1.13.0</td>
</tr>
<tr>
<td>I2C Interface</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>I2S Interface</td>
<td>1.0.0</td>
<td>v1.9.0</td>
</tr>
<tr>
<td>I3C Interface</td>
<td>0.1.0</td>
<td>v3.2.0</td>
</tr>
<tr>
<td>I3C Address-related Helper Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3C Common Command Codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3C Devicetree related bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3C In-Band Interrupts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3C Target Device API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3C Transfer API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPM Interface</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Input Interface</td>
<td>0.1.0</td>
<td>v3.4.0</td>
</tr>
<tr>
<td>Input Event Definitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inter-VM Shared Memory (ivshmem) reference API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keyboard Matrix API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keyboard Scan Driver APIs</td>
<td>1.0.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>LED Interface</td>
<td>1.0.0</td>
<td>v1.12.0</td>
</tr>
<tr>
<td>LED Strip Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoRa APIs</td>
<td>0.1.0</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>MBOX Interface</td>
<td>0.1.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>MIPI Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPI Display interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPI-DBI driver APIs</td>
<td>0.1.0</td>
<td>v3.6.0</td>
</tr>
<tr>
<td>MIPI-DSI driver APIs</td>
<td>0.1.0</td>
<td>v3.1.0</td>
</tr>
<tr>
<td>MODBUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous Drivers APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devmux Driver APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT8xx driver APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT8xx co-processor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT8xx common functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT8xx display list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT8xx memory map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT8xx reference API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi Function Device Drivers APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MFD ADS592 interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MFD AXP192 interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MFD BD8LB600FS interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MFD NPM1300 Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCI Express Controller Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCle Host Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCle Capabilities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCle Host MSI Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCle Host PTM Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCle Virtual Channel Host Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PECI Interface</td>
<td>1.0.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>PS/2 Driver APIs</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>PWM Interface</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Pin Controller Interface</td>
<td>0.1.0</td>
<td>v3.0.0</td>
</tr>
<tr>
<td>Dynamic Pin Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTC DS3231 Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTC Interface</td>
<td>0.1.0</td>
<td>v3.4.0</td>
</tr>
</tbody>
</table>
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regulator Interface</td>
<td>0.1.0</td>
<td>v2.4.0</td>
</tr>
<tr>
<td>ADP5360 Devicetree helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AXP192 Devicetree helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX20335 Devicetree helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NPM1100 Devicetree helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NPM1300 Devicetree helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NPM6001 Devicetree helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regulator Parent Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCA9420 Utilities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reset Controller Interface</td>
<td>0.1.0</td>
<td>v3.1.0</td>
</tr>
<tr>
<td>Retained memory driver interface</td>
<td>0.8.0</td>
<td>v3.4.0</td>
</tr>
<tr>
<td>SDHC interface</td>
<td>0.1.0</td>
<td>v3.1.0</td>
</tr>
<tr>
<td>SMBus Interface</td>
<td>0.1.0</td>
<td>v3.4.0</td>
</tr>
<tr>
<td>SPI Interface</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>SYSCON Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensor Interface</td>
<td>1.0.0</td>
<td>v1.2.0</td>
</tr>
<tr>
<td>Sensor emulator backend API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text Display Interface</td>
<td>0.1.0</td>
<td>v3.4.0</td>
</tr>
<tr>
<td>Time-aware GPIO Interface</td>
<td>0.1.0</td>
<td>v3.5.0</td>
</tr>
<tr>
<td>UART Interface</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Async UART API</td>
<td>0.8.0</td>
<td>v1.14.0</td>
</tr>
<tr>
<td>Interrupt-driven UART API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polling UART API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UART Mux Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Power Delivery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Type-C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB Type-C Port Controller API</td>
<td>0.1.0</td>
<td>v3.1.0</td>
</tr>
<tr>
<td>USB device controller driver API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB host controller driver API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB-C VBUS API</td>
<td>0.1.0</td>
<td>v3.3.0</td>
</tr>
<tr>
<td>Video Controls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video Interface</td>
<td>1.0.0</td>
<td>v2.1.0</td>
</tr>
<tr>
<td>Video pixel formats</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watchdog Interface</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Device Model</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>Device memory-mapped IO management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Named MMIO region macros</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single MMIO region macros</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top-level MMIO region macros</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree</td>
<td>1.0.0</td>
<td>v2.2.0</td>
</tr>
<tr>
<td>“For-each” macros</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bus helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chosen nodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependency tracking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree CAN API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree Clocks API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree DMA API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree Fixed Partition API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree GPIO API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree IO Channels API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree MBOX API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree PWMs API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree Reset Controller API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Devicetree SPI API</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page

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

continues on next page
<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-Off Service APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipe APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semaphore APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spinlock APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Clock APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread Stack APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timer APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Mode APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User mode mutex APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User mode semaphore APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Work Queue APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory heaps based on memory attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory-Attr Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modem APIs</td>
<td>0.1.0</td>
<td>v3.5.0</td>
</tr>
<tr>
<td>Modem CMUX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modem PPP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modem Pipe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modem Ubx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System Services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Console API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coredump APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crypto</td>
<td>1.0.0</td>
<td>v1.7.0</td>
</tr>
<tr>
<td>Cipher</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Function APIs</td>
<td>1.0.0</td>
<td>v1.0.0</td>
</tr>
<tr>
<td>File System APIs</td>
<td>1.0.0</td>
<td>v1.5.0</td>
</tr>
<tr>
<td>File System Storage</td>
<td>1.0.0</td>
<td>v1.11.0</td>
</tr>
<tr>
<td>Flash Circular Buffer (FCB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash Circular Buffer Data Structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcb API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fcb non-API prototypes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-volatile Storage (NVS)</td>
<td>1.0.0</td>
<td>v1.12.0</td>
</tr>
<tr>
<td>Non-volatile Storage APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-volatile Storage Data Structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Settings</td>
<td>1.0.0</td>
<td>v1.12.0</td>
</tr>
<tr>
<td>Settings backend interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Settings name processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Settings subsystem runtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash image API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Listener APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared multi-heap interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC service APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC service RPMsg API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC service backend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC service static VRINGs API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Icmsg IPC library API</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Icmsg multi-endpoint IPC library API</td>
<td>0.1.0</td>
<td>v3.5.0</td>
</tr>
<tr>
<td>Packed Buffer API</td>
<td>0.1.0</td>
<td>v3.5.0</td>
</tr>
<tr>
<td>RPMsg service APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterable Sections APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linkable loadable extensions</td>
<td>1.0.0</td>
<td>v1.13.0</td>
</tr>
<tr>
<td>ELF data types and defines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLEXT symbols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linkable loadable extensions buffer loader</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loader context for llext</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logging</td>
<td>1.0.0</td>
<td>v1.13.0</td>
</tr>
<tr>
<td>Logger system</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log link API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log message API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log output API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log output formatting flags.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logger backend interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logger multidomain backend helpers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logger backend standard interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logger control API</td>
<td></td>
<td>v1.13.0</td>
</tr>
<tr>
<td>Logging API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr callback API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr fs_mgmt callback API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr img_mgmt callback API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr os_mgmt callback API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr settings_mgmt callback API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr handler API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr img_mgmt API</td>
<td>1.0.0</td>
<td>v1.11.0</td>
</tr>
<tr>
<td>MCUmgr img_mgmt_client API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr mgmt API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr os_mgmt_client API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCUmgr transport SMP API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMP client API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Banks Driver APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Blocks APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Management Driver APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power Management (PM)</td>
<td></td>
<td>v1.2.0</td>
</tr>
<tr>
<td>CPU Power Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Runtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2RAM APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>States</td>
<td></td>
<td>v1.2.0</td>
</tr>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hooks</td>
<td></td>
<td>v1.2.0</td>
</tr>
<tr>
<td>Policy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTIO</td>
<td>0.1.0</td>
<td>v3.2.0</td>
</tr>
<tr>
<td>RTIO CQE Flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTIO MPSC API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTIO Priorities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTIO SPSC API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTIO SQE Flags</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retention API</td>
<td>0.1.0</td>
<td>v3.4.0</td>
</tr>
<tr>
<td>Boot mode interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bootloader info interface</td>
<td>0.1.0</td>
<td>v3.5.0</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>API</th>
<th>Version</th>
<th>Available in Zephyr Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semihosting APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shell API</td>
<td>1.0.0</td>
<td>v1.14.0</td>
</tr>
<tr>
<td>State Machine Framework API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disk Access Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream to flash interface</td>
<td>0.1.0</td>
<td>v2.3.0</td>
</tr>
<tr>
<td>flash area Interface</td>
<td>1.0.0</td>
<td>v1.11.0</td>
</tr>
<tr>
<td>System Initialization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System power off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task Watchdog APIs</td>
<td>0.8.0</td>
<td>v2.5.0</td>
</tr>
<tr>
<td>Thread analyzer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timing Measurement APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arch specific Timing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Board specific Timing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SoC specific Timing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object tracking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional Variable Tracing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFO Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIFO Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mailbox Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Slab Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Queue Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutex Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM Device Runtime Tracing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipe Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poll Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semaphore Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Syscall Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System PM Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timer Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Work Delayable Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Work Poll Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Work Queue Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Work Tracing APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracing format APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracing utility macros</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zbus APIs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensing Sensor API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensor Callbacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensing Subsystem API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensor Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Emulator interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2C Emulation Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPI Emulation Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eSPI Emulation Interface</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

continues on next page

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

continues on next page
2.6.2 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.

![API Life Cycle](image)

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)

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

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

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 Architecture meeting where, barring any objections, the Pull Request will be merged

When the API changes status to stable API, mark the API version in the headers where the API is defined. Stable APIs shall have a version where the major version is one or larger (x.y.z \( x \geq 1 \)). (see api overview <api_overview>)
Introducing breaking API 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).

A breaking API 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 a breaking API 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: Breaking 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 Breaking API Change label

3. The RFC issue must be submitted for discussion in the next Zephyr Architecture 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 Architecture 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

2.6. API Status and Guidelines
The labels API, Breaking API Change and Release Notes, as well as any others that are applicable

The label Architecture Review if the RFC was not yet discussed and agreed upon in Zephyr Architecture meeting

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.

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

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

### 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 Architecture 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.

2.6.3 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 `const 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:

  ```c
  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:
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`:

```c
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)
2.6.4 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 `DEVICEDEFINE()` or `SYSINIT()` 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.

## 2.7  Language Support

### 2.7.1  C Language Support

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

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

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

**Language Standards**

Zephyr does not target a specific version of the C standards; however, the Zephyr codebase makes extensive use of the features newly introduced in the 1999 release of the ISO C standard (ISO/IEC 9899:1999, hereinafter referred to as C99) such as those listed below, effectively requiring the use of a compiler toolchain that supports the C99 standard and above:
• inline functions
• standard boolean types (bool in <stdbool.h>)
• fixed-width integer types ([u]intN_t in <stdint.h>)
• designated initializers
• variadic macros
• restrict qualification

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

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

Standard Library

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

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

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

Dynamic Memory Management  The common dynamic memory management implementa-
tion can be enabled by selecting the CONFIG_COMMON_LIBC_MALLOC in the application configura-
tion file.

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

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

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

There are also separate controls to select calloc() (COMMON_LIBC_CALLOC) and reallocarray()
(COMMON_LIBC_REALLOCARRAY). Both of these are enabled by default as that doesn’t impact mem-
ory usage in applications not using them.
The standard dynamic memory management interface functions implemented by the common C library are thread safe and may be simultaneously called by multiple threads. These functions are implemented in `lib/libc/common/source/stdlib/malloc.c`.

**Minimal libc** The most basic C library, named “minimal libc”, is part of the Zephyr codebase and provides the minimal subset of the standard C library required to meet the needs of Zephyr and its subsystems, 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. It can also be used with a number of different toolchains.

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

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

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

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

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

**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 defined in the POSIX `errno.h` specification and other de-facto standard sources have been added to the minimal libc.

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

ENOEEXEC
   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.

EXIST
   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.

ENOBUFFS
   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.

ECNNREUSED
   Connection refused.

EADDRINUSE
   Address already in use.

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.

Newlib  Newlib is a complete C library implementation written for the embedded systems. It is a separate open source project and is not included in source code form with Zephyr. Instead, the Zephyr SDK includes a precompiled library for each supported architecture (lib.a and libm.a).
Zephyr Project Documentation, Release 3.6.99

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For every release of Zephyr, the toolchain-bundled Picolibc and the Picolibc module are guaranteed to be in sync when using the recommended version of Zephyr SDK.
Building Without Toolchain bundled Picolibc  For toolchain where there is no bundled Picolibc, it is still possible to use Picolibc by building it from source. Note that any restrictions mentioned in Picolibc Module still apply.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dynamic Memory Management

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

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

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

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

Note: Native Zephyr applications should use the memory management API supported by the Zephyr kernel such as `k_malloc()` in order to take advantage of the advanced features that they offer.

C standard dynamic memory management interface functions such as `malloc()` should be used only by the portable applications and libraries that target multiple operating systems.

2.7.2 C++ Language Support

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

Enabling C++ Support

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

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

The default C++ standard level (i.e. the language enforced by the compiler flags passed) for Zephyr apps is C++11. Other standards are available via kconfig choice, for example `CONFIG_STD_CPP98`. The oldest standard supported and tested in Zephyr is C++98.
When compiling a source file, the build system selects the C++ compiler based on the suffix (extension) of the files. Files identified with either a `cpp` or a `cxx` suffix are compiled using the C++ compiler. For example, `myCplusplusApp.cpp` is compiled using C++.

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

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

### Language Features

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

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

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

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

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.

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

### Zephyr Minimal C++ Library

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

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

The scope of the minimal C++ library is strictly limited to providing the basic C++ language support, and it does not implement any Standard Template Library (STL) classes and functions. For this reason, it is only suitable for use in the applications that implement their own (non-standard) class library and do not rely on the Standard Template Library (STL) components.

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

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

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

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

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

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

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

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

2.8 Optimizations

Guides on how to optimize Zephyr for performance, power and footprint.

2.8.1 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:
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.

### 2.8.2 Optimization Tools

The available optimization tools let you analyse *Footprint and Memory Usage* and *Data Structures* using different build system targets.

**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.

Some of the tools mentioned in this section are organizing their output based on the physical organization of the symbols. As some symbols might be external to the project's tree structure, or might lack metadata needed to display them by name, the following top-level containers are used to group such symbols:

- **Hidden** - The RAM and ROM reports list all processing symbols with no matching mapped files in the Hidden category.
  
  This means that the file for the listed symbol was not added to the metadata file, was empty, or was undefined. The tool was unable to get the name of the function for the given symbol nor identify where it comes from.

- **No paths** - The RAM and ROM reports list all processing symbols with relative paths in the No paths category.
  
  This means that the listed symbols cannot be placed in the tree structure of the report at an absolute path under one specific file. The tool was able to get the name of the function, but it was unable to identify where it comes from.

**Note:** You can have multiple cases of the same function, and the No paths category will list the sum of these in one entry.
**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, as in the following example.

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 -Bbuild -GNinja -DBOARD=reel_board samples/hello_world
# Now run the build tool on the generated build system:
ninja -Cbuild ram_report
```

These commands will generate something similar to the output below:

<table>
<thead>
<tr>
<th>Path</th>
<th>Size</th>
<th>%</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>4637</td>
<td>100.00%</td>
<td>-</td>
</tr>
<tr>
<td>(hidden)</td>
<td>4</td>
<td>0.09%</td>
<td>-</td>
</tr>
<tr>
<td>(no paths)</td>
<td>2748</td>
<td>59.26%</td>
<td>-</td>
</tr>
<tr>
<td>├── _cpus_active</td>
<td>4</td>
<td>0.09%</td>
<td>0x20000314</td>
</tr>
<tr>
<td>│ └── _kernel</td>
<td>32</td>
<td>0.69%</td>
<td>0x20000318</td>
</tr>
<tr>
<td>│ └── _sw_isr_table</td>
<td>384</td>
<td>8.28%</td>
<td>0x00006474</td>
</tr>
<tr>
<td>│ └── cli.1</td>
<td>16</td>
<td>0.35%</td>
<td>0x20000254</td>
</tr>
<tr>
<td>│ └── _kernel</td>
<td>4</td>
<td>0.09%</td>
<td>0x20000264</td>
</tr>
<tr>
<td>│ └── _sw_isr_table</td>
<td>384</td>
<td>8.28%</td>
<td>0x00006474</td>
</tr>
<tr>
<td>│ └── on.2</td>
<td>128</td>
<td>2.76%</td>
<td>0x20000120</td>
</tr>
<tr>
<td>│ └── poll_out_lock.0</td>
<td>128</td>
<td>2.76%</td>
<td>0x20000120</td>
</tr>
<tr>
<td>│ └── z_idle_threads</td>
<td>128</td>
<td>2.76%</td>
<td>0x20000120</td>
</tr>
<tr>
<td>│ └── z_interrupt_stacks</td>
<td>2048</td>
<td>44.17%</td>
<td>0x20000360</td>
</tr>
<tr>
<td>│ └── _kernel</td>
<td>4</td>
<td>0.09%</td>
<td>0x200002d4</td>
</tr>
<tr>
<td>│ └── _sw_isr_table</td>
<td>384</td>
<td>8.28%</td>
<td>0x00006474</td>
</tr>
<tr>
<td>│ └── cli.1</td>
<td>16</td>
<td>0.35%</td>
<td>0x20000254</td>
</tr>
<tr>
<td>│ └── _kernel</td>
<td>4</td>
<td>0.09%</td>
<td>0x20000264</td>
</tr>
<tr>
<td>│ └── poll_out_lock.0</td>
<td>128</td>
<td>2.76%</td>
<td>0x20000120</td>
</tr>
<tr>
<td>│ └── _kernel</td>
<td>4</td>
<td>0.09%</td>
<td>0x200002d4</td>
</tr>
<tr>
<td>├── WORKSPACE</td>
<td>184</td>
<td>3.97%</td>
<td>-</td>
</tr>
<tr>
<td>│ └── modules</td>
<td>184</td>
<td>3.97%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ └── nordic</td>
<td>184</td>
<td>3.97%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ └── hal</td>
<td>184</td>
<td>3.97%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ └── drivers</td>
<td>184</td>
<td>3.97%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ └── sources</td>
<td>184</td>
<td>3.97%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── nrfx_clock.c</td>
<td>8</td>
<td>0.17%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ │ └── m_clock cb</td>
<td>8</td>
<td>0.17%</td>
<td>0x200002e4</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── nrfx_gpiote.c</td>
<td>132</td>
<td>2.85%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ │ └── m cb</td>
<td>132</td>
<td>2.85%</td>
<td>0x20000060</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── nrfx_ppi.c</td>
<td>4</td>
<td>0.09%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── m_channels_allocated</td>
<td>4</td>
<td>0.09%</td>
<td>0x20000e4</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── nrfx_twim.c</td>
<td>40</td>
<td>0.86%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── m cb</td>
<td>40</td>
<td>0.86%</td>
<td>0x200002e4</td>
</tr>
<tr>
<td>│ │ └── ZEPHYR_BASE</td>
<td>1701</td>
<td>36.68%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ └── arch</td>
<td>5</td>
<td>0.11%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ └── arm</td>
<td>5</td>
<td>0.11%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ └── core</td>
<td>5</td>
<td>0.11%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── mpu</td>
<td>1</td>
<td>0.02%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── static_regions_num</td>
<td>1</td>
<td>0.02%</td>
<td>0x2000348</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── tls.c</td>
<td>4</td>
<td>0.09%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ │ └── z_arm_tls_ptr</td>
<td>4</td>
<td>0.09%</td>
<td>0x2000240</td>
</tr>
<tr>
<td>│ │ │ │ └── drivers</td>
<td>258</td>
<td>5.56%</td>
<td>-</td>
</tr>
<tr>
<td>│ │ │ │ │ └── ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

```
4637
```
**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` target with your board, as in the following example.

Using west:

```
wst 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 -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

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

These commands will generate something similar to the output below:

<table>
<thead>
<tr>
<th>Path</th>
<th>Size</th>
<th>%</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>27828</td>
<td>100.00%</td>
<td></td>
</tr>
<tr>
<td>ZEPHYR_BASE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arch</td>
<td>13558</td>
<td>48.72%</td>
<td></td>
</tr>
<tr>
<td>cortex_m</td>
<td>1020</td>
<td>3.67%</td>
<td></td>
</tr>
<tr>
<td>fault.c</td>
<td>620</td>
<td>2.23%</td>
<td></td>
</tr>
<tr>
<td>bus_fault.constprop.0</td>
<td>108</td>
<td>0.39%</td>
<td>0x00000749</td>
</tr>
<tr>
<td>mem_manage_fault.constprop.0</td>
<td>120</td>
<td>0.43%</td>
<td>0x000007b5</td>
</tr>
<tr>
<td>usage_fault.constprop.0</td>
<td>84</td>
<td>0.30%</td>
<td>0x000006f5</td>
</tr>
<tr>
<td>z_arm_fault</td>
<td>292</td>
<td>1.05%</td>
<td>0x0000082d</td>
</tr>
<tr>
<td>z_arm_fault_init</td>
<td>16</td>
<td>0.66%</td>
<td>0x00000951</td>
</tr>
<tr>
<td>boards</td>
<td>32</td>
<td>0.11%</td>
<td></td>
</tr>
<tr>
<td>reel_board</td>
<td>32</td>
<td>0.11%</td>
<td></td>
</tr>
<tr>
<td>board.c</td>
<td>32</td>
<td>0.11%</td>
<td></td>
</tr>
<tr>
<td>__init_board_reel_board_init</td>
<td>8</td>
<td>0.03%</td>
<td>0x0000063e4</td>
</tr>
<tr>
<td>board_reel_board_init</td>
<td>24</td>
<td>0.09%</td>
<td>0x00000ed5</td>
</tr>
<tr>
<td>build</td>
<td>194</td>
<td>0.70%</td>
<td></td>
</tr>
<tr>
<td>zephyr</td>
<td>194</td>
<td>0.70%</td>
<td></td>
</tr>
<tr>
<td>isr_tables.c</td>
<td>192</td>
<td>0.69%</td>
<td></td>
</tr>
<tr>
<td>___irq_vector_table</td>
<td>192</td>
<td>0.69%</td>
<td>0x00000040</td>
</tr>
<tr>
<td>misc</td>
<td>2</td>
<td>0.01%</td>
<td></td>
</tr>
<tr>
<td>generated</td>
<td>2</td>
<td>0.01%</td>
<td></td>
</tr>
<tr>
<td>configs.c</td>
<td>2</td>
<td>0.01%</td>
<td></td>
</tr>
<tr>
<td>__ConfigAbsSyms</td>
<td>2</td>
<td>0.01%</td>
<td>0x00005945</td>
</tr>
<tr>
<td>drivers</td>
<td>6282</td>
<td>22.57%</td>
<td></td>
</tr>
</tbody>
</table>

21652

**Build Target: puncover**  This target uses a third-party tool called puncover which can be found at [https://github.com/HBehrens/puncover](https://github.com/HBehrens/puncover). 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, install the puncover Python module:
pip3 install git+https://github.com/HBehrens/puncover --user

**Warning:** This is a third-party tool that might or might not be working at any given time. Please check the GitHub issues, and report new problems to the project maintainer.

After you installed the Python module, use puncover target with your board, as in the following example.

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 build system:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world
# Now run the build tool on the generated build system:
ninja -Cbuild 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 build system:
cmake -Bbuild -GNinja -DBOARD=reel_board -DCONFIG_STACK_USAGE=y samples/hello_world
# Now run the build tool on the generated build system:
ninja -Cbuild puncover
```

**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
```

Alternatively, you can get it from fedora:

```
sudo dnf install dwarves
```

After you installed the package, use pahole target with your board, as in the following example.

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:
```bash
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world
```
# Now run the build tool on the generated build system:
```bash
ninja -Cbuild pahole
```

Pahole will generate something similar to the output below in the console:

```c
/* 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 */
```

## 2.9 Flashing and Hardware Debugging

### 2.9.1 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>;
   }
}(continues on next page)```
A typical flash layout for devices with a ROM bootloader and storage partition is:

```
/*
* 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";
    reg = <0x0003c000 0x00004000>;
};
```

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 by 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)

### Enabling BOSSAC on Windows Native [Experimental]

Zephyr SDK’s `bossac` is currently supported on Linux and macOS only. Windows support can be achieved by using the `bossac` version from BOSSA official releases. After installing using default options, the `bossac.exe` must be added to Windows PATH. A specific `bossac` executable can be used by passing the `--bossac` option, as follows:

```
west flash -r bossac --bossac="C:\Program Files (x86)\BOSSA\bossac.exe" --bossac-port="COMx"
```

**Note:** WSL is not currently supported.

### LinkServer Debug Host Tools

Linkserver is a utility for launching and managing GDB servers for NXP debug probes, which also provides a command-line target flash programming capabilities. Linkserver can be used with the [NXP MCUXpresso for Visual Studio Code](https://mcuxpresso.xilinx.com) implementation, with custom debug configurations based on GNU tools or as part of a headless solution for continuous integration and test. LinkServer can be used with MCU-Link, LPC-Link2, LPC11U35-based and OpenSDA based standalone or on-board debug probes from NXP.

NXP recommends installing LinkServer by using NXP’s [MCUXpresso Installer](https://mcuxpresso.xilinx.com). This method will also install the tools supporting the debug probes below, including NXP’s MCU-Link and LPC-Scrypt tools.

LinkServer is compatible with the following debug probes:

- `LPC-LINK2 CMSIS DAP Onboard Debug Probe`
- `MCU-Link CMSIS-DAP Onboard Debug Probe`
- `OpenSDA DAPLink Onboard Debug Probe`

To use LinkServer with West commands, the install folder should be added to the `PATH environment variable`. The default installation path to add is:

Linux

```
/usr/local/LinkServer
```

Windows

```
c:\nxp\LinkServer_<version>
```

Supported west commands:

1. `flash`
2. `debug`
3. `debugserver`
4. attach

Notes:

1. Probes can be listed with LinkServer:

   ```
   LinkServer probes
   ```

2. With multiple debug probes attached to the host, use the LinkServer west runner `--probe` option to pass the probe index.

   ```
   west flash --runner=linkserver --probe=3
   ```

3. Device-specific settings can be overridden with the west runner for LinkServer with the option `--override`. May be used multiple times. The format is dictated by LinkServer, e.g.:

   ```
   west flash --runner=linkserver --override /device/memory/5/flash-driver=MIMXRT500_SFDP_MXIC_OSPI_S.cfx
   ```

### 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*
- *MCU-Link JLink 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:

- LPC-LINK2 CMSIS DAP Onboard Debug Probe
- MCU-Link CMSIS-DAP Onboard Debug Probe
- OpenSDA DAPLink Onboard Debug Probe
- ST-LINK/V2-1 Onboard Debug Probe

Check if your SoC is listed in pyOCD Supported Devices.

**Lauterbach TRACE32 Debug Host Tools**

Lauterbach TRACE32 is a product line of microprocessor development tools, debuggers and real-time tracer with support for ARM Cortex-A/-R/-M, RISC-V, Xtensa, etc. Zephyr allows users to develop and program boards with Lauterbach TRACE32 support using `west`.

The runner consists of a wrapper around TRACE32 software, and allows a Zephyr board to execute a custom start-up script (Practice Script) for the different commands supported, including the ability to pass extra arguments from CMake. It is up to the board using this runner to define the actions performed on each command.

**Install Lauterbach TRACE32 Software**

[Download Lauterbach TRACE32 software](#) and follow the installation steps described in Lauterbach TRACE32 Installation Guide.

**Flashing and Debugging**

Set the `environment variable` `T32_DIR` to the TRACE32 system directory. Then execute `west flash` or `west debug` commands to flash or debug the Zephyr application as detailed in *Building, Flashing and Debugging*. The debug command launches TRACE32 GUI to allow debug the Zephyr application, while the flash command hides the GUI and perform all operations in the background.

By default, the `t32` runner will launch TRACE32 using the default configuration file named `config.t32` located in the TRACE32 system directory. To use a different configuration file, supply the argument `--config CONFIG` to the runner, for example:

```
west flash --config myconfig.t32
```

For more options, run `west flash --context -r t32` to print the usage.
Zephyr RTOS Awareness  To enable Zephyr RTOS awareness follow the steps described in Lauterbach TRACE32 Zephyr OS Awareness Manual.

NXP S32 Debug Probe Host Tools

NXP S32 Debug Probe is designed to work in conjunction with NXP S32 Design Studio for S32 Platform.

Download (registration required) NXP S32 Design Studio for S32 Platform and follow the S32 Design Studio for S32 Platform Installation User Guide to get the necessary debug host tools and associated USB device drivers.

Note that Zephyr RTOS-awareness support for the NXP S32 GDB server depends on the target device. Consult the product release notes for more information.

Supported west commands:

1. debug
2. debugserver
3. attach

Basic usage  Before starting, add NXP S32 Design Studio installation directory to the system PATH environment variable. Alternatively, it can be passed to the runner on each invocation via --s32ds-path as shown below:

Linux

west debug --s32ds-path=/opt/NXP/S32DS.3.5

Windows

west debug --s32ds-path=C:\\NXP\S32DS.3.5

If multiple S32 debug probes are connected to the host via USB, the runner will ask the user to select one via command line prompt before continuing. The connection string for the probe can be also specified when invoking the runner via --dev-id=<connection-string>. Consult NXP S32 debug probe user manual for details on how to construct the connection string. For example, if using a probe with serial ID 00:04:9f:00:ca:fe:

west debug --dev-id='s32dbg:00:04:9f:00:ca:fe'

It is possible to pass extra options to the debug host tools via --tool-opt. When executing debug or attach commands, the tool options will be passed to the GDB client only. When executing debugserver, the tool options will be passed to the GDB server. For example, to load a Zephyr application to SRAM and afterwards detach the debug session:

west debug --tool-opt='--batch'

2.9.2  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.

2.9.  Flashing and Hardware Debugging
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

<table>
<thead>
<tr>
<th>Debug Probes</th>
<th>Host Tools</th>
<th>J-Link Debug</th>
<th>OpenOCD</th>
<th>pyOCD</th>
<th>NXP S32DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPC-Link2</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSDA</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DAPLink</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSDA</td>
<td>J-Link</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>J-Link</td>
<td>J-Link External</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ST-LINK/V2-1</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>some STM32 boards</td>
</tr>
<tr>
<td>NXP S32 Debug Probe</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

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.

### MCU-Link CMSIS-DAP Onboard Debug Probe

The CMSIS-DAP debug probes allow debugging from any compatible toolchain, including IAR EWARM, Keil MDK, NXP’s MCUXpresso IDE and MCUXpresso extension for VS Code. In addition to debug probe functionality, the MCU-Link probes may also provide:

1. SWO trace end point: this virtual device is used by MCUXpresso to retrieve SWO trace data. See the MCUXpresso IDE documentation for more information.
2. Virtual COM (VCOM) port / UART bridge connected to the target processor
3. USB to UART, SPI and/or I2C interfaces (depending on MCU-Link type/implementation)
4. Energy measurements of the target MCU

This debug probe is compatible with the following debug host tools:

- LinkServer Debug Host Tools
This probe is realized by programming the MCU-Link microcontroller with the CMSIS-DAP MCU-Link firmware, which is already installed by default. NXP recommends using NXP's MCUXpresso Installer, which installs both the MCU-Link host tools plus the LinkServer Debug Host Tools.

1. Put the MCU-Link microcontroller into DFU boot mode by attaching the DFU jumper, then powering up the board.
2. Run the program_CMSIS script, found in the installed MCU-Link scripts folder.
3. Remove the DFU jumper and power cycle the board.

**MCU-Link J-Link Onboard Debug Probe**

The MCU-Link J-Link is an onboard debug probe and usb-to-serial adapter supported on many NXP development boards.

This debug probe is compatible with the following debug host tools:

- **J-Link Debug Host Tools**

These probes do not have JLink firmware installed by default, and must be updated. NXP recommends using NXP's MCUXpresso Installer, which installs both the J-Link Debug Host Tools plus the MCU-Link host tools.

1. Put the MCU-Link microcontroller into DFU boot mode by attaching the DFU jumper, then powering up the board.
2. Run the program_JLINK script, found in the installed MCU-Link scripts folder.
3. Remove the DFU jumper and power cycle the board.

**LPC-LINK2 CMSIS DAP Onboard Debug Probe**

The CMSIS-DAP debug probes allow debugging from any compatible toolchain, including IAR EWARM, Keil MDK, as well as NXP's MCUXpresso IDE and MCUXpresso extension for VS Code. As well as providing debug probe functionality, the LPC-Link2 probes also provide:

1. SWO trace end point: this virtual device is used by MCUXpresso to retrieve SWO trace data. See the MCUXpresso IDE documentation for more information.
2. Virtual COM (VCOM) port / UART bridge connected to the target processor
3. LPCSIO bridge that provides communication to I2C and SPI slave devices

This probe is realized by programming the LPC-Link2 microcontroller with the CMSIS-DAP 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_CMSIS script.
3. Remove the DFU jumper and power cycle the board.

**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 **LPCScript** 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 information 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:

```bash
# From the root of the zephyr repository
west build -b None samples/hello_world
west flash
```

2.9. Flashing and Hardware Debugging
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.

```cmake
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.

```bash
west flash --runner jlink
```

To attach a debugger to your board and open up a debug console with jlink.

```bash
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:

```bash
$ telnet localhost 19021
```

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 menuconfig

**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

```bash
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).

**NXP S32 Debug Probe**

NXP S32 Debug Probe enables NXP S32 target system debugging via a standard debug port while connected to a developer's workstation via USB or remotely via Ethernet.
NXP S32 Debug Probe is designed to work in conjunction with NXP S32 Design Studio (S32DS) and NXP Automotive microcontrollers and processors. Install the debug host tools as indicated in NXP S32 Debug Probe Host Tools before you program the firmware.

2.10 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 workflow 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

Additionally, in some cases modules (particularly vendor HALs) can contain references to optional binary blobs.

This page summarizes a list of policies and best practices which aim at better organizing the workflow in Zephyr modules.

2.10.1 Modules vs west projects

Zephyr modules, described in this page, are not the same as west projects. In fact, modules do not require west at all. However, when using modules with west, then the build system uses west in order to find modules.

In summary:

Modules are repositories that contain a zephyr/module.yml file, so that the Zephyr build system can pull in the source code from the repository. West projects are entries in the projects: section in the west.yml manifest file. West projects are often also modules, but not always. There are west projects that are not included in the final firmware image (eg. tools) and thus do not need to be modules. Modules are found by the Zephyr build system either via west itself, or via the ZEPHYR_MODULES CMake variable.

The contents of this page only apply to modules, and not to west projects in general (unless they are a module themselves).
2.10.2 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.

- Module repositories names should be explicitly set in the zephyr/module.yml file.
- 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’.

- Modules should expose all provided header files with an include pathname beginning with the module-name. (E.g., mcuboot should expose its bootutil/bootutil.h as “mcuboot/bootutil/bootutil.h”.)

### Synchronizing with upstream

It is preferred to synchronize a module repository 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 suppressing 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.

2.10.3 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.
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*. 

---

**Zephyr Project Documentation, Release 3.6.99**

**Note:** Module owners are not required to be Zephyr Maintainers.
2.10.4 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.

2.10.5 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.

2.10.6 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 inte-
grated 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 plat-
forms, 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.

2.10.7 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.

2.10.8 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 describ-
ing 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 EXTRA_ZEPHYR_MODULES CMake variable or 
by adding a EXTRA_ZEPHYR_MODULES line to .zephyrrc (See the section on Environment 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 -DEXTRA_ZEPHYR_MODULES=/ 
<path>/foo then the module given by the command line variable EXTRA_ZEPHYR_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.

If you want to permanently add modules to the zephyr workspace and you are using zephyr as your manifest repository, you can also add a west manifest file into the submanifests directory. See submanifests/README.txt for more details.

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.

## 2.10.9 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 should be specified in the zephyr/module.yml file. This will ensure the module name is not changeable through user-defined directory names or west manifest files:

```yaml
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:

```yaml
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:

```yaml
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:

```yaml
build:
  cmake: .
  kconfig: Kconfig
```

**Sysbuild integration**

*Sysbuild* is the Zephyr build system that allows for building multiple images as part of a single application, the sysbuild build process can be extended externally with modules as needed, for example to add custom build steps or add additional targets to a build. Inclusion of sysbuild-specific build files, CMakeLists.txt and Kconfig, can be described as:

```yaml
build:
  sysbuild-cmake: <cmake-directory>
  sysbuild-kconfig: <directory>/Kconfig
```

The `sysbuild-cmake: <cmake-directory>` part specifies that `<cmake-directory>` contains the CMakeLists.txt to use. The `sysbuild-kconfig: <directory>/Kconfig` part specifies the Kconfig file to use.

Here is an example module.yml file referring to CMakeLists.txt and Kconfig files in the `sysbuild` directory of the module:

```yaml
build:
  sysbuild-cmake: sysbuild
  sysbuild-kconfig: sysbuild/Kconfig
```

The module description file `zephyr/module.yml` can also 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:

```yaml
build:
  sysbuild-cmake-ext: True
  sysbuild-kconfig-ext: True
```

This allows control of the build inclusion to be described externally to the Zephyr module.

**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.

**Zephyr modules** 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:

```cmake
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:

```kconfig
source "$(ZEPHYR_FOO_MODULE_DIR)/some/Kconfig"
```

During CMake processing of each Zephyr module, the following variables are also available:

- the current module's name: `${ZEPHYR_CURRENT_MODULE_NAME}`
- the current module's top level directory: `${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:

```cmake
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:

```cmake
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.

**Sysbuild modules**  In both Kconfig and CMake, the variable SYSBUILD_CURRENT_MODULE_DIR contains the absolute path to the sysbuild module. In CMake, SYSBUILD_CURRENT_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 sysbuild module:

- In CMake: use `${SYSBUILD_CURRENT_MODULE_DIR}` for the module's top level directory, and `${SYSBUILD_CURRENT_CMAKE_DIR}` for the directory containing its CMakeLists.txt
- In Kconfig: use `${SYSBUILD_CURRENT_MODULE_DIR}` for the module's top level directory

In Kconfig, the variable may be used to find additional files to include. For example, to include the file some/Kconfig:

```kconfig
source "$(SYSBUILD_CURRENT_MODULE_DIR)/some/Kconfig"
```

The module can source additional CMake files using these variables. For example:

```cmake
include(${SYSBUILD_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:
Sysbuild modules hooks Sysbuild provides an infrastructure which allows a sysbuild module to define a function which will be invoked by sysbuild at a pre-defined point in the CMake flow.

Functions invoked by sysbuild:

- `<module-name>_pre_cmake(IMAGES <images>)`: This function is called for each sysbuild module before CMake configure is invoked for all images.
- `<module-name>_post_cmake(IMAGES <images>)`: This function is called for each sysbuild module after CMake configure has completed for all images.
- `<module-name>_pre_domains(IMAGES <images>)`: This function is called for each sysbuild module before domains yaml is created by sysbuild.
- `<module-name>_post_domains(IMAGES <images>)`: This function is called for each sysbuild module after domains yaml has been created by sysbuild.

Arguments passed from sysbuild to the function defined by a module:

- `<images>` is the list of Zephyr images that will be created by the build system.

If a module `foo` want to provide a post CMakem configure function, then the module’s sysbuild CMakeLists.txt file must define function `foo_post_cmake()`.

To facilitate naming of functions, the module name is provided by sysbuild CMake through the `SYSBUILD_CURRENT_MODULE_NAME` CMake variable when loading the module’s sysbuild CMakeLists.txt file.

Example of how the `foo` sysbuild module can define `foo_post_cmake()`:

```cmake
function(${SYSBUILD_CURRENT_MODULE_NAME}_post_cmake)
  cmake_parse_arguments(POST_CMAKE "" "" IMAGES ${ARGN})
  message("Invoking ${CMAKE_CURRENT_FUNCTION}. Images: ${POST_CMAKE_IMAGES}"")
endfunction()
```

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.

```cmake
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:

```cmake
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 MOD-ULE_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 FOO module.

Create the following structure

```
<MODULE_EXT_ROOT>
├── modules
│   └── foo
│       ├── CMakeLists.txt
│       └── Kconfig
```

and inside the modules.cmake file, add the following content
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.
- **snippet_root**: Contains additional snippets that are available for use. These snippets must be defined in snippet.yml files underneath the `<snippet_root>/snippets` folder. For example, if you have snippet_root: foo, then you should place your module's snippet.yml files in `<your-module>/foo/snippets` or any nested subdirectory.
- **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).
- **sca_root**: Contains additional SCA tool implementations available to the build system. Each tool must be located in `<sca_root>/sca/<tool>` folder. The folder must contain a sca.cmake.

Example of a module.yml 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:
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
```

Binary Blobs

Zephyr supports fetching and using *binary blobs*, and their metadata is contained entirely in `zephyr/module.yml`. This is because a binary blob must always be associated with a Zephyr module, and thus the blob metadata belongs in the module’s description itself.

Binary blobs are fetched using `west blobs`. If `west` is *not used*, they must be downloaded and verified manually.

The `blobs` section in `zephyr/module.yml` consists of a sequence of maps, each of which has the following entries:

- **path**: The path to the binary blob, relative to the `zephyr/blobs/` folder in the module repository
- **sha256**: SHA-256 checksum of the binary blob file
- **type**: The *type of binary blob*. Currently limited to `img` or `lib`
- **version**: A version string
- **license-path**: Path to the license file for this blob, relative to the root of the module repository
- **url**: URL that identifies the location the blob will be fetched from, as well as the fetching scheme to use
- **description**: Human-readable description of the binary blob
- **doc-url**: A URL pointing to the location of the official documentation for this blob

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.

### 2.10.10 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. Drafts reduce noise by not automatically notifying anyone until marked as “Ready for review”. 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:

```yaml
- 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:

```yaml
- 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 and expectations.

2. Submit a pull request changing the entry referencing the module into the `west.yml` of the main Zephyr tree with the following information:

```yaml
- 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:

```yaml
- 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.
2.11  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:

\[
\text{west [common-opts] <command> [opts] <args>}
\]

Since west v0.8, you can also run west like this:

\[
\text{python3 -m west [common-opts] <command> [opts] <args>}
\]

You can run \texttt{west --help} (or \texttt{west -h} for short) to get top-level help for available west commands, and \texttt{west <command> -h} for detailed help on each command.

2.11.1  Installing west

West is written in Python 3 and distributed through PyPI. Use pip3 to install or upgrade west:

On Linux:

\[
\text{pip3 install --user -U west}
\]

On Windows and macOS:

\[
\text{pip3 install -U west}
\]

\textbf{Note:}  See Python and pip for additional clarification on using the \texttt{--user} switch.

Afterwards, you can run \texttt{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 \textit{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 commands like \texttt{west init}, \texttt{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.

\(^1\) Zephyr is an English name for the Latin Zephyrus, the ancient Greek god of the west wind.
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:

```bash
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`:
  ```bash
  brew install bash-completion
  ```
  then source the main bash completion script in your `~/.bash_profile`:
  ```bash
  source /usr/local/etc/profile.d/bash_completion.sh
  ```
  and finally copy `west-completion.bash` to `/usr/local/etc/bash_completion.d/`.

2.11.2 West Release Notes

v1.2.0

Major changes:

- New `west grep` command for running a “grep tool” in your west workspace’s repositories. Currently, `git grep`, `ripgrep`, and standard `grep` are supported grep tools.
  
  To run this command to get `git grep foo` results from all cloned, active repositories, run:
  ```bash
  west grep foo
  ```

  Here are some other examples for running different grep commands with `west grep`:

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>git grep --untracked</code></td>
<td><code>west grep --untracked foo</code></td>
</tr>
<tr>
<td><code>ripgrep</code></td>
<td><code>west grep --tool ripgrep foo</code></td>
</tr>
<tr>
<td><code>grep --recursive</code></td>
<td><code>west grep --tool grep foo</code></td>
</tr>
</tbody>
</table>
To switch the default grep tool in your workspace, run the appropriate command in this table:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ripgrep</td>
<td>west config grep.tool ripgrep</td>
</tr>
<tr>
<td>grep</td>
<td>west config grep.tool grep</td>
</tr>
</tbody>
</table>

For more details, run `west help grep`.

Other changes:
- The manifest file format now supports a `description` field in each `projects:` element. See `Projects` for examples.
- `west list --format` now accepts `{description}` in the format string, which prints the project's `description: value`.
- `west compare` now always prints information about the `manifest.rev branch`.

Bug fixes:
- `west init` aborts if the destination directory already exists.

API changes:
- `west.commands.WestCommand` methods `check_call()` and `check_output()` now take any kwargs that can be passed on to the underlying subprocess function.
- `west.commands.WestCommand.run_subprocess()`: new wrapper around `subprocess.run()`. This could not be named `run()` because `WestCommand` already had a method by this name.
- `west.commands.WestCommand` methods `dbg()`, `inf()`, `wrn()`, and `err()` now all take an `end` kwarg, which is passed on to the call to `print()`.
- `west.manifest.Project` now has a `description` attribute, which contains the parsed value of the `description:` field in the manifest data.

v1.1.0

Major changes:
- `west compare`: new command that compares the state of the workspace against the manifest.
- Support for a new `manifest.project-filter` configuration option. See `Built-in Configuration Options` for details. The `west manifest --freeze` and `west manifest --resolve` commands currently cannot be used when this option is set. This restriction can be removed in a later release.
- Project names which contain comma (,) or whitespace now generate warnings. These warnings are errors if the new `manifest.project-filter` configuration option is set. The warnings may be promoted to errors in a future major version of west.

Other changes:
- `west forall` now takes a `--group` argument that can be used to restrict the command to only run in one or more groups. Run `west help forall` for details.
- All `west` commands will now output log messages from `west` API modules at warning level or higher. In addition, the `--verbose` argument to `west` can be used once to include informational messages, or twice to include debug messages, from all commands.

Bug fixes:
- Various improvements to error messages, debug logging, and error handling.

API changes:
• `west.manifest.Manifest.is_active()` now respects the `manifest.project-filter` configuration option's value.

v1.0.1

Major changes:
• Manifest schema version “1.0” is now available for use in this release. This is identical to the “0.13” schema version in terms of features, but can be used by applications that do not wish to use a “0.x” manifest “version:” field. See Version for details on this feature.

Bug fixes:
• West no longer exits with a successful error code when sent an interrupt signal. Instead, it exits with a platform-specific error code and signals to the calling environment that the process was interrupted.

v1.0.0

Major changes in this release:
• The west-apis are now declared stable. Any breaking changes will be communicated by a major version bump from v1.x.y to v2.x.y.
• West v1.0 no longer works with the Zephyr v1.14 LTS releases. This LTS has long been obsoleted by Zephyr v2.7 LTS. If you need to use Zephyr v1.14, you must use west v0.14 or earlier.
• Like the rest of Zephyr, west now requires Python v3.8 or later
• West commands no longer accept abbreviated command line arguments. For example, you must now specify `west update --keep-descendants` instead of using an abbreviation like `west update --keep-d`. This is part of a change applied to all of Zephyr's Python scripts' command-line interfaces. The abbreviations were causing problems in practice when commands were updated to add new options with similar names but different behavior to existing ones.

Other changes:
• All built-in west functions have stopped using `west.log`
• `west update`: new `--submodule-init-config` option. See commit 9ba92b05 for details.

Bug fixes:
• West extension commands that failed to load properly sometimes dumped stack. This has been fixed and west now prints a sensible error message in this case.
• `west config` now fails on malformed configuration option arguments which lack a . in the option name

API changes:
• The west package now contains the metadata files necessary for some static analyzers (such as `mypy`) to auto-detect its type annotations. See commit d9f00e24 for details.
• the deprecated `west.build` module used for Zephyr v1.14 LTS compatibility was removed
• the deprecated `west.cmake` module used for Zephyr v1.14 LTS compatibility was removed
• the `west.log` module is now deprecated. This module uses global state, which can make it awkward to use it as an API which multiple different python modules may rely on.

2.11. West (Zephyr's meta-tool)
• The west-apis-commands module got some new APIs which lay groundwork for a future change to add a global verbosity control to a command’s output, and work to remove global state from the west package’s API:
  – New `west.commands.WestCommand.__init__()` keyword argument: `verbosity`
  – New `west.commands.WestCommand` property: `color_ui`
  – New `west.commands.WestCommand` methods, which should be used to print output from extension commands instead of writing directly to `sys.stdout` or `sys.stderr`: `inf()`, `wrn()`, `err()`, `die()`, `banner()`, `small_banner()`
  – New `west.commands.VERBOSITY` enum

v0.14.0

Bug fixes:
• West commands that were run with a bad local configuration file dumped stack in a confusing way. This has been fixed and west now prints a sensible error message in this case.
• A bug in the way west looks for the zephyr repository was fixed. The bug itself usually appeared when running an extension command like `west build` in a new workspace for the first time; this used to fail (just for the first time, not on subsequent command invocations) unless you ran the command in the workspace’s top level directory.
• West now prints sensible error messages when the user lacks permission to open the manifest file instead of dumping stack traces.

API changes:
• The `west.manifest.MalformedConfig` exception type has been moved to the `west.configuration` module
• The `west.manifest.MalformedConfig` exception type has been moved to the `west.configuration` module
• The `west.configuration.Configuration` class now raises `MalformedConfig` instead of `RuntimeError` in some cases

v0.13.1

Bug fix:
• When calling `west.manifest.Manifest.from_file()` when outside of a workspace, west again falls back on the `ZEPHYR_BASE` environment variable to locate the workspace.

v0.13.0

New features:
• You can now associate arbitrary user data with the manifest repository itself in the manifest: `self: userdata: value, like so:

```yaml
manifest:
  self:
    userdata: <any YAML value can go here>
```

Bug fixes:
• The path to the manifest repository reported by `west` could be incorrect in certain circumstances detailed in [issue #572](https://github.com/zephyrproject-rtos/west/issues/572). This has been fixed as part of a larger overhaul of path handling support in the `west.manifest` API module.

• The `west.Manifest.ManifestProject.__repr__` return value was fixed

API changes:

• `west.configuration.Configuration`: new object-oriented interface to the current configuration. This reflects the system, global, and workspace-local configuration values, and allows you to read, write, and delete configuration options from any or all of these locations.

• `west.commands.WestCommand`:
  – `config`: new attribute, returns a `Configuration` object or aborts the program if none is set. This is always usable from within extension command `do_run()` implementations.
  – `has_config`: new boolean attribute, which is True if and only if reading `self.config` will abort the program.

• The path handling in the `west.manifest` package has been overhauled in a backwards-incompatible way. For more details, see commit [56cfe8d1d1](https://github.com/zephyrproject-rtos/west/commit/56cfe8d1d1f3c9b45de3e793c738acd62db52aca).

• `west.manifest.Manifest.validate()`: this now returns the validated data as a Python dict. This can be useful if the value passed to this function was a str, and the dict is desired.

• `west.manifest.Manifest`: new:
  – `path` attributes: `abspath`, `posixpath`, `relative_path`, `yaml_path`, `repo_path`, `repo_posixpath`
  – `userdata` attribute, which contains the parsed value from manifest: `self: userdata;`, or is None
  – `from_topdir()` factory method

• `west.manifest.ManifestProject`: new `userdata` attribute, which also contains the parsed value from manifest: `self: userdata;`, or is None

• `west.manifest.ManifestImportFailed`: the constructor can now take any value; this can be used to reflect failed imports from a `map` or other compound value.

• Deprecated configuration APIs:

The following APIs are now deprecated in favor of using a `Configuration` object. Usually this will be done via `self.config` from a `WestCommand` instance, but this can be done directly by instantiating a `Configuration` object for other usages.

  – `west.configuration.config`
  – `west.configuration.read_config`
  – `west.configuration.update_config`
  – `west.configuration.delete_config`

v0.12.0

New features:

• `West` now works on the MSYS2 platform.

• West manifest files can now contain arbitrary user data associated with each project. See `Repository user data` for details.
Bug fixes:

- The `west list` command's `{sha}` format key has been fixed for the manifest repository; it now prints `N/A` (“not applicable”) as expected.

API changes:

- The `west.manifest.Project.userdata` attribute was added to support project user data.

**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. In-
stead, 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 values
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 informa-
tion.
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*. 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 it 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

    (continues on next page)
• 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` kwarg 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.

2.11.3 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:
The `git fetch` command example in the last line above is what needs to succeed.

One strategy is to go to `~/path/to/your/project`, 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:
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:

```bash
$ 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.):

```bash
$ 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:

```bash
$ 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)
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.

2.11.4 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:
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, zcbor 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). Note how-
ever that modules and projects are conceptually different.

**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 man-
aged 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.

2.11.5 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:

```
w 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

```
w west update [-f {always,smart}] [-k] [-r] [-g GROUP] [--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 Git¹.

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 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** 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`.

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.
**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 compare`: compare the state of the workspace against the manifest
- `west diff`: run `git diff` in local project repositories
- `west forall`: run an arbitrary command in local project repositories
- `west grep`: search for patterns in local project repositories
- `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 status`: run `git status` 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

2.11.6 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:

2.11. West (Zephyr’s meta-tool) 139
To use this helper on just the projects foo and bar, run:

```bash
west forall -c "git config credential.helper store" foo bar
```

To use this helper by default on your computer, run:

```bash
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.)

You can use the Git credential store to authenticate with a GitHub PAT (Personal Access Token) like so:

```bash
echo "https://x-access-token:$GH_TOKEN@github.com" >> ~/.git-credentials
```

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.

If you setup fetching via SSH, you can use Git URL rewrite feature. The following command instructs Git to use SSH URLs for GitHub instead of HTTPS ones:

```bash
git config --global url."git@github.com:".insteadOf "https://github.com/"
```

**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  
  │    └── src/  
  │        └── main.c  
  │        └── west.yml # main manifest with optional import(s) and override(s)  
  └── modules/  
      └── lib/  
          └── zcbor/ # .git/ project from either the main manifest or some import.  
  └── zephyr/  
      └── west.yml # .git/ project  
```

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:
```

(continues on next page)
(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
│   └── 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/
│       └── zcbor/ # .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
```

(continues on next page)
You can also do this “by hand” by copy/pasting zephyr/west.yml as shown above for the T2 topology, with the same caveats.

2.11.7 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. 3: 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 → B, as do P1 and P3 from F → G.
  - P3 moves forward from A → B.
  - P3 moves backward from C → 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 → 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:
# ... same remotes as above ...
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>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Mandatory; a unique name for the remote.</td>
</tr>
<tr>
<td>url-base</td>
<td>A prefix that is prepended to the fetch URL for each project with this remote.</td>
</tr>
</tbody>
</table>

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. Note that west projects are different from modules.

Here is an example. We'll assume the remotes given above.

```manifest:
# [... same remotes as above ...]
projects:
- name: proj1
  description: the first example project
  remote: remote1
  path: extra/project-1
- name: proj2
  description: A multi-line description of the second example project.
  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 3: projects elements keys

<table>
<thead>
<tr>
<th>Key(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>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.</td>
</tr>
<tr>
<td>description</td>
<td>Optional, an informational description of the project. Added in west v1.2.0.</td>
</tr>
<tr>
<td>remote, url</td>
<td>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 project's remote. Otherwise, the manifest is invalid.</td>
</tr>
<tr>
<td>repo-path</td>
<td>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.</td>
</tr>
<tr>
<td>revision</td>
<td>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. Using HEAD~0 as the revision will cause west to keep the current state of the project.</td>
</tr>
<tr>
<td>path</td>
<td>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.</td>
</tr>
<tr>
<td>clone-depth</td>
<td>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.</td>
</tr>
<tr>
<td>west-commands</td>
<td>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.</td>
</tr>
<tr>
<td>import</td>
<td>Optional. If true, imports projects from manifest files in the given repository into the current manifest. See Manifest Imports for details.</td>
</tr>
<tr>
<td>groups</td>
<td>Optional, a list of groups the project belongs to. See Project Groups for details.</td>
</tr>
<tr>
<td>submodules</td>
<td>Optional. You can use this to make west update also update Git submodules defined by the project. See Git Submodules in Projects for details.</td>
</tr>
<tr>
<td>userdata</td>
<td>Optional. The value is an arbitrary YAML value. See Repository user data.</td>
</tr>
</tbody>
</table>

**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:

```yaml
manifest:
  defaults:
    remote: remote1
    revision: v1.3

remotes:
  - name: remote1
```

---

1 In git, HEAD is a reference, whereas HEAD~<n> is a valid revision but not a reference. West fetches references, such as refs/heads/main or HEAD, and commits not available locally, but will not fetch commits if they are already available. HEAD~0 is resolved to a specific commit that is locally available, and therefore west will simply checkout the locally available commit, identified by HEAD~0.

2.11. West (Zephyr’s meta-tool)
url-base: https://git.example.com/base1
  - name: remote2
    url-base: https://git.example.com/base2

projects:
  - name: proj1
    description: the first example project
    path: extra/project-1
    revision: master
  - name: proj2
    description: A multi-line description of the second example project.
    repo-path: my-path
    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>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remote</td>
<td>Optional. This will be used for a project's remote if it does not have a url or remote key set.</td>
</tr>
<tr>
<td>revision</td>
<td>Optional. This will be used for a project's revision if it does not have one set. If not given, the default is master.</td>
</tr>
</tbody>
</table>

**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`:

```yaml
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 5: self keys

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>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 <a href="https://git.example.com/project-repo">https://git.example.com/project-repo</a>, the manifest repository would be cloned to the directory project-repo.</td>
</tr>
<tr>
<td>west-commands</td>
<td>Optional. This is analogous to the same key in a project sequence element.</td>
</tr>
<tr>
<td>import</td>
<td>Optional. This is also analogous to the projects key, but allows importing projects from other files in the manifest repository. See Manifest Imports.</td>
</tr>
</tbody>
</table>

### Version

The version subsection declares that the manifest file uses features which were introduced in some version of west. Attempts to load the manifest with older versions of west will fail with an error message that explains the minimum required version of west which is needed.

Here is an example:

```yaml
manifest:
  # Marks that this file uses version 0.10 of the west manifest file format.
  # An attempt to load this manifest file with west v0.8.0 will fail with an error message saying that west v0.10.0 or later is required.
  version: "0.10"
```

The pykwalify schema `manifest-schema.yml` in the west source code repository is used to validate the manifest section.

Here is a table with the valid version values, along with information about the manifest file features that were introduced in that version.
<table>
<thead>
<tr>
<th>version</th>
<th>New features</th>
</tr>
</thead>
<tbody>
<tr>
<td>“0.7”</td>
<td>Initial support for the version feature. All manifest file features that are not otherwise mentioned in this table were introduced in west v0.7.0 or earlier.</td>
</tr>
<tr>
<td>“0.8”</td>
<td>Support for import: path-prefix: <em>(Option 3: Mapping)</em></td>
</tr>
<tr>
<td>“0.9”</td>
<td><strong>Use of west v0.9.x is discouraged.</strong> This schema version is provided to allow users to explicitly request compatibility with west v0.9.0. However, west v0.10.0 and later have incompatible behavior for features that were introduced in west v0.9.0. You should ignore version “0.9” if possible.</td>
</tr>
</tbody>
</table>
| “0.10” | Support for:  
- submodules: in projects: *(Git Submodules in Projects)*  
- manifest: group-filter:, and groups: in projects: *(Project Groups)*  
- The import: feature now supports allowlist: and blocklist:; these are respectively recommended as replacements for older names as part of a general Zephyr-wide inclusive language change. The older key names are still supported for backwards compatibility. *(Manifest Imports, Option 3: Mapping)* |
| “0.12” | Support for userdata: in projects: *(Repository user data)* |
| “0.13” | Support for self: userdata: *(Repository user data)* |
| “1.0”  | Identical to “0.13”, but available for use by users that do not wish to use a “0.x” version field. |
| “1.2”  | Support for description: in projects: *(Projects)* |

**Note:** Versions of west without any new features in the manifest file format do not change the list of valid version values. For example, version: “0.11” is **not** valid, because west v0.11.x did not introduce new manifest file format features.

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.

If you do not include a version in your manifest, each new release of west assumes that it should try to load it using the features that were available in that release. This may result in error messages that are harder to understand if that version of west is too old to load the manifest.

**Group-filter**  
See Project Groups.

**Active and Inactive Projects**

Projects defined in the west manifest can be **inactive** or **active**. The difference is that an inactive project is generally ignored by west. For example, west update will not update inactive projects, and west list will not print information about them by default. As another example, any Manifest Imports in an inactive project will be ignored by west.

There are two ways to make a project inactive:

1. Using the manifest.project-filter configuration option. If a project is made active or inactive using this option, then the rules related to making a project inactive using its groups: are ignored. That is, if a regular expression in manifest.project-filter applies to a project, the project’s groups have no effect on whether it is active or inactive.

See the entry for this option in Built-in Configuration Options for details.
2. Otherwise, if a project has groups, and they are all disabled, then the project is inactive. See the following section for details.

**Project Groups**

You can use the `groups:` and `group-filter:` keys briefly described above to place projects into groups, and to enable or disable groups.

For example, this lets you run a `west forall` command only on the projects in the group by using `west forall --group`. This can also let you make projects inactive; see the previous section for more information on inactive projects.

The next section introduces project groups. The following section describes *Enabled and Disabled Project Groups*. 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.

**Groups Basics**

The `groups:` and `group-filter:` keys appear in the manifest like this:

```manifest:
projects:
  - name: some-project
    groups: ...
    group-filter: ...
```

The `groups:` key's value is a list of group names. Group names are strings.

You can enable or disable project groups using `group-filter:`. Projects whose groups are all disabled, and which are not otherwise made active by a manifest `project-filter` configuration option, are inactive.

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 is necessary to avoid some pathological edge cases.)
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.

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.

**Note:** In all of the examples that follow, the `manifest.project-filter` option is assumed to be unset.

Example 1: no disabled groups

The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
```

(continues on next page)
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:

```yaml
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

  group-filter: [-groupA]
```

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:

```yaml
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB
    - name: baz

  group-filter: [-groupA,-groupB]
```

(continues on next page)
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]

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:
  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:

```yaml
# 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:

```yaml
# 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:

```yaml
# parent/west.yml:
manifest:
  group-filter: [+unstable,-optional]
  projects:
    - name: child
```

(continues on next page)
And child/west.yml contains:

```yaml
# 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:

```yaml
# 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:

```yaml
# 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:

<table>
<thead>
<tr>
<th>manifest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>projects:</td>
</tr>
<tr>
<td>- name: some-project</td>
</tr>
<tr>
<td>submodules: ...</td>
</tr>
</tbody>
</table>

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 in sync, along with another project named bar in the same workspace.

You can do that with this manifest file:

<table>
<thead>
<tr>
<th>manifest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>projects:</td>
</tr>
<tr>
<td>- name: foo</td>
</tr>
<tr>
<td>submodules: true</td>
</tr>
<tr>
<td>- name: bar</td>
</tr>
</tbody>
</table>

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:
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.

**Repository user data**

West versions v0.12 and later support an optional `userdata` key in projects.

West versions v0.13 and later supports this key in the manifest: `self:` section.

It is meant for consumption by programs that require user-specific project metadata. Beyond parsing it as YAML, west itself ignores the value completely.

The key's value is arbitrary YAML. West parses the value and makes it accessible to programs using `west-apis` as the `userdata` attribute of the corresponding `west.manifest.Project` object.

Example manifest fragment:

```yaml
manifest:
  projects:
    - name: foo
      submodules:
        - path: path/to/foo-first-sub
        - name: foo-second-sub
          path: path/to/foo-second-sub
        - name: bar
```

Example Python usage:

```python
manifest = west.manifest.Manifest.from_file()
foo, bar, baz = manifest.get_projects(['foo', 'bar', 'baz'])

foo.userdata  # None
bar.userdata  # 'a-string'
baz.userdata  # {'key': 'value'}
manifest.userdata  # 'blub'
```

**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:

```yaml
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**
  - Example 1.1: Downstream of a Zephyr release
  - Example 1.2: “Rolling release” Zephyr downstream
  - Example 1.3: Downstream of a Zephyr release, with module fork

- **Relative path**
  - Example 2.1: Downstream of a Zephyr release with explicit path
  - Example 2.2: Downstream with directory of manifest files
  - Example 2.3: Continuous Integration overrides

- **Mapping with additional configuration**
  - Example 3.1: Downstream with name allowlist
  - Example 3.2: Downstream with path allowlist
  - Example 3.3: Downstream with path blocklist
  - Example 3.4: Import into a subdirectory

- **Sequence of paths and mappings**
  - Example 4.1: Downstream with sequence of manifest files
  - Example 4.2: Import order illustration

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 re-solving 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:

```yaml
manifest:
  # ...
  projects:
    - name: p1
      revision: v1.0
      import: true  # Import west.yml from p1's v1.0 git tag
    - name: p2
```

(continues on next page)
It's an error to set `import` to either `true` or `false` inside `self`, like this:

```
manifest:
  # ...
  self:  # Error
    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
  │   ├── 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`:

```
wst 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.

2.11. West (Zephyr's meta-tool)
You can create the workspace in the same way:

```bash
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

```yaml
# 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
# imported projects have lower precedence
```
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:

```yaml
# 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`.

```yaml
# 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

# downstream west.yml:
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
```

An equivalent manifest in a single file would be:

```yaml
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`.

```yaml
# mainline west.yml:
manifest:
  projects:
    - name: app
```

(continues on next page)
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
      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 blacklist 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
      path: libraries/lib
      name: lib
      path: libraries/lib
      name: lib2
      path: libraries/lib2
    - name: hal_foo
      path: modules/hals/foo  # excluded

(continues on next page)
- name: hal_bar
  path: modules/hals/bar  # excluded
- name: hal_baz
  path: modules/hals/baz  # excluded

# downstream west.yml:
manifest:
  projects:
  - name: mainline
    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
  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/

(continues on next page)
You can do this using this manifest:

```yaml
manifest:
  projects:
    - name: foo
      url: https://git.example.com/foo
      import:
        path-prefix: external-code
```

An equivalent manifest in a single file would be:

```yaml
# 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.

```yaml
# 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:

```yaml
# my-repo/west.yml
manifest:
  # ...
  projects:
    - name: my-library
```

(continues on next page)
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:

```yaml
# 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 west-commands keys 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.

The following sections describe these outcomes.

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 filterN + ... + filter-2 + filter-1 + 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.

Here, “invalid” means that the syntax of the manifest file doesn’t follow the rules documented on this page.

If your manifest is valid but it’s not working the way you want it to, turning up the verbosity with `-v` is a good way to get detailed information about what decisions west made about your manifest, and why:

```
west -v manifest --validate
```

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.

2.11.8 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):

```
wst config zephyr.base
```

You can switch to another zephyr repository without changing manifest.path – and thus the behavior of commands like west update – using:

```
wst 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:

```
wst config --global color.ui false
```

To undo the above change:

```
wst 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.
### 2.11. West (Zephyr’s meta-tool)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color.ui</td>
<td>Boolean. If true (the default), then west output is colorized when stdout is a terminal.</td>
</tr>
<tr>
<td>commands.allow Extensions</td>
<td>Boolean, default true, disables Extensions if false</td>
</tr>
<tr>
<td>grep.color</td>
<td>String, default empty. Set this to never to disable west grep color output. If set, west grep passes the value to the grep tool’s --color option.</td>
</tr>
<tr>
<td>grep.tool</td>
<td>String, one of &quot;git-grep&quot; (default), &quot;ripgrep&quot;, or &quot;grep&quot;. The grep tool that west grep should use.</td>
</tr>
<tr>
<td>grep.&lt;TOOL&gt;-args</td>
<td>String, default empty. The &lt;TOOL&gt; part is a pattern that can be any grep.tool value, so grep.ripgrep-args is an example configuration option. If set, arguments that west grep should pass to the corresponding grep tool. Run west help grep for details.</td>
</tr>
<tr>
<td>grep.&lt;TOOL&gt;-path</td>
<td>String, default empty. The &lt;TOOL&gt; part is a pattern that can be any grep.tool value, so grep.ripgrep-path is an example configuration option. The path to the corresponding tool that west grep should use instead of searching for the command. Run west help grep for details.</td>
</tr>
<tr>
<td>manifest.file</td>
<td>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.</td>
</tr>
<tr>
<td>manifest.group-filter</td>
<td>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 &quot;+foo,-bar&quot; enables group foo and disables bar. See Project Groups.</td>
</tr>
<tr>
<td>manifest.path</td>
<td>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.</td>
</tr>
<tr>
<td>manifest.project-filter</td>
<td>Comma-separated list of strings. The option’s value is a comma-separated list of regular expressions, each prefixed with + or -, like this:</td>
</tr>
</tbody>
</table>

```
+re1,-re2,-re3
```

Project names are matched against each regular expression (re1, re2, re3,...) in the list, in order. If the entire project name matches the regular expression, that element of the list either deactivates or activates the project. The project is deactivated if the element begins with -. The project is activated if the element begins with +. (Project names cannot contain, if this option is used, so the regular expressions do not need to contain a literal , character).

If a project's name matches multiple regular expressions in the list, the result from the last regular expression is used. For example, if manifest.project-filter is:

```
-hal_*,+hal_foo
```

Then a project named hal_bar is inactive, but a project named hal_foo is active.

If a project is made inactive or active by a list element, the project is active or not regardless of whether any or all of its groups are disabled. (This is currently the only way to make a project that has no groups inactive.)

Otherwise, i.e. if a project does not match any regular expressions in the list, it is active or inactive according to the usual rules related to its groups (see Project Group Examples for examples in that case).

Within an element of a manifest.project-filter list, leading and trailing whitespace are ignored. That means these example values are equivalent:

```
+foo,-bar
+foo , -bar
```

Any empty elements are ignored. That means these example values are equivalent:
2.11.9 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
- 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.

176 Chapter 2. Developing with Zephyr
'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:
        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:
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:
  - name: my-command-name
class: MyCommand
help: one-line help for what my-command-name does

  - name: command2
class: 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-commands.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.

### 2.11.10 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`.

2.11. West (Zephyr's meta-tool)
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).

Sysbuild (multi-domain builds) Sysbuild (System build) can be used to create a multi-domain build system combining multiple images for a single or multiple boards.

Use --sysbuild to select the Sysbuild (System build) build infrastructure with west build to build multiple domains.

More detailed information regarding the use of sysbuild can be found in the Sysbuild (System build) guide.

Tip: The build.sysbuild configuration option can be enabled to tell west build to default build using sysbuild. --no-sysbuild can be used to disable sysbuild for a specific build.

west build will build all domains through the top-level build folder of the domains specified by sysbuild.

A single domain from a multi-domain project can be built by using --domain argument.

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:

```bash
west build -c
```

**Setting a Default Board** To configure west build to build for the reel_board by default:

```bash
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:

```bash
west build -b <BOARD> path/to/source/directory
```

To set the build directory explicitly, use --build-dir (or -d):

```bash
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:

```bash
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` makes no attempt to detect if the build directory needs to be made pristine. This can lead to errors if you do something like try to reuse a build directory for a different --board.

Using `--pristine=auto` makes `west build` detect some of these situations and make the build directory pristine before trying the build.

**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`:

```
wst -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 the CMake build configuration step, even if a build system has already been generated. This will make incremental builds slower (but still much faster than building from scratch).

After using `--` once to generate the build directory, use `west build -d <build-dir>` on subsequent runs to do incremental builds.
Alternatively, make your CMake arguments permanent as described in the next section; it will not slow down 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`:

```
w west build -- -DEXTRA_CONF_FILE=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 to make sure it will take effect.

For example, to always enable `CMAKE_EXPORT_COMPILE_COMMANDS`, you can run:

```
w 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:

```
w 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`:

```
w 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:

```cmake
set(CMAKE_EXPORT_COMPILE_COMMANDS ON CACHE BOOL "")
set(CMAKE_VERBOSE_MAKEFILE ON CACHE BOOL "")
```

Then run:

```
w west config build.cmake-args -C ~/my-cache.cmake
```
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:

```
w west build -o=-j4
```

The `-o` option is described further in the previous section.

**Build a single domain**  In a multi-domain build with `hello_world` and `MCUboot`, you can use `--domain hello_world` to only build this domain:

```
w west build --sysbuild --domain hello_world
```

The `--domain` argument can be combined with the `--target` argument to build the specific target for the target, for example:

```
w west build --sysbuild --domain hello_world --target help
```

**Use a snippet**  See *Using Snippets*.

**Configuration Options**  You can `configure` `west build` using these options.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build.board</td>
<td>String. If given, this the board used by west build when --board is not given and BOARD is unset in the environment.</td>
</tr>
<tr>
<td>build.board_warn</td>
<td>Boolean, default true. If false, disables warnings when west build can't figure out the target board.</td>
</tr>
<tr>
<td>build.cmake-args</td>
<td>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.</td>
</tr>
<tr>
<td>build.dir-fmt</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• board: The board name</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• app: The name of the source directory.</td>
</tr>
<tr>
<td>build.generator</td>
<td>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)</td>
</tr>
<tr>
<td>build.guess-dir</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• never (default): Never try to guess, bail out instead and require the user to provide a build folder with -d.</td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td>build.pristine</td>
<td>String. Controls the way in which west build may clean the build folder before building. Can take the following values:</td>
</tr>
<tr>
<td></td>
<td>• never (default): Never automatically make the build folder pristine.</td>
</tr>
<tr>
<td></td>
<td>• 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).</td>
</tr>
<tr>
<td></td>
<td>• always: Always make the build folder pristine before building, if a build system is present.</td>
</tr>
<tr>
<td>build.sysbuild</td>
<td>Boolean, default false. If true, build application using the sysbuild infrastructure.</td>
</tr>
</tbody>
</table>

**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:

```bash
# 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:

```bash
west flash --H -r jlink
```

**Multi-domain flashing**  When a Sysbuild (multi-domain builds) folder is detected, then `west flash` will flash all domains in the order defined by sysbuild.

It is possible to flash the image from a single domain in a multi-domain project by using `--domain`. For example, in a multi-domain build with `hello_world` and MCUboot, you can use the `--domain hello_world` domain to only flash the only image from this domain:

```bash
west flash --domain hello_world
```

**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):

```bash
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):

```bash
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`):

```bash
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:

```bash
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:

```
wест 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`):

```
wест 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:

```
wест debug -H -r jlink
```

Multi-domain debugging  `west debug` can only debug a single domain at a time. When a `Sysbuild (multi-domain builds)` folder is detected, `west debug` will debug the default domain specified by `sysbuild`.

The default domain will be the application given as the source directory. See the following example:

```
wест build --sysbuild path/to/source/directory
```

For example, when building `hello_world` with `MCUboot` using `sysbuild`, `hello_world` becomes the default domain:

```
wест build --sysbuild samples/hello_world
```

So to debug `hello_world` you can do:

```
wест debug
```

or:

```
wест debug --domain hello_world
```

If you wish to debug `MCUboot`, you must explicitly specify `MCUboot` as the domain to debug:

```
wест debug --domain MCUboot
```
**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.

```python
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.

class runners.core.DeprecatedAction(option_strings, dest, nargs=None, const=None,
    default=None, type=None, choices=None, required=False, help=None, metavar=None)

class runners.core.FileType(value, names=None, *, module=None, qualname=None,
    type=None, start=1, boundary=None)
```
exception runners.core.MissingProgram(program)
    FileNotFoundError subclass for missing program dependencies.

    No significant changes from the parent FileNotFoundError; this is useful for explicitly sig-
    naling 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: ~typing.Set[str] = <factory>,
    dev_id: bool = False, flash_addr: bool = False, erase: bool = False, reset: bool =
    False, tool_opt: bool = False, file: 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’, ‘at-
      tach’.

    • dev_id: whether the runner supports device identifiers, in the form of an -i, –dev-id
      option. This is useful when the user has multiple debuggers connected to a single com-
      puter, in order to select which one will be used with the command provided.

    • 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.)

    • reset: whether the runner supports a –reset option, which resets the device after a
      flash operation is complete.

    • tool_opt: whether the runner supports a –tool-opt (-O) option, which can be given mul-
      tiple times and is passed on to the underlying tool that the runner wraps.

    • file: whether the runner supports a –file option, which specifies exactly the file that
      should be used to flash, overriding any default discovered in the build directory.

class runners.core.RunnerConfig(build_dir: str, board_dir: str, elf_file: str | None, exe_file: str
    | None, hex_file: str | None, bin_file: str | None, uf2_file: str
    | None, file: str | None, file_type: FileType | None = FileType.OTHER, gdb: str | None = None, openocd: str |
    None = 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.

```python
bin_file: str | None
    Alias for field number 5
board_dir: str
    Alias for field number 1
build_dir: str
    Alias for field number 0
elf_file: str | None
    Alias for field number 2
exe_file: str | None
    Alias for field number 3
file: str | None
    Alias for field number 7
file_type: FileType | None
    Alias for field number 8
gdb: str | None
    Alias for field number 9
hex_file: str | None
    Alias for field number 4
openocd: str | None
    Alias for field number 10
openocd_search: List[str]
    Alias for field number 11
uf2_file: str | None
    Alias for field number 6
```

```python
class runners.core.ZephyrBinaryRunner(cfg: 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() class-method, 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.

```python
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.

```python
property build_conf: BuildConfiguration
```

Get a BuildConfiguration for the build directory.

```python
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() → 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: RunnerConfig, args: Namespace) → 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().

classmethod dev_id_help() → str

    Get the ArgParse help text for the –dev-id option.

abstract classmethod do_add_parser(parser)

    Hook for adding runner-specific options.

abstract classmethod do_create(cfg: RunnerConfig, args: Namespace) → 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: BuildConfiguration)

    If CONFIG_HAS_FLASH_LOAD_OFFSET is n in build_conf, return
    the CONFIG_FLASH_BASE_ADDRESS value. Otherwise, return CON-  
    FIG_FLASH_BASE_ADDRESS + CONFIG_FLASH_LOAD_OFFSET.

static get_flash_address(args: Namespace, build_conf: BuildConfiguration, default: int = 0) → int

    Helper method for extracting a flash address.
    If args.dt_flash is true, returns the address obtained from ZephyrBinaryRun-  
    ner.flash_address_from_build_conf(build_conf).
    Otherwise (when args.dt_flash is False), the default value is returned.
static get_runners() → List[Type[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], **kwargs) → Popen
Spawn a child command, ensuring it ignores SIGINT.
The returned subprocess.Popen object must be manually terminated.

static require(program: str, path: str | None = None) → str
Require that a program is installed before proceeding.

Parameters

- **program** – name of the program that is required, or path to a program
  binary.
- **path** – PATH where to search for the program binary. By default check
  on the system PATH.

If program is an absolute path to an existing program binary, this call succeeds. Other-
wise, try to find the program by name on the system PATH or in the given PATH, if
provided.

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, **kwargs)
Run a client that handles SIGINT.

run_server_and_client(server, client, **kwargs)
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.
- the keyword arguments, if any, will be passed down to both server and client sub-
  process calls

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.

classmethod tool_opt_help() → str
Get the ArgParse help text for the –tool-opt option.

2.11. West (Zephyr's meta-tool) 193
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).

2.11.11 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. In some configurations, `west sign` is also used to invoke an external, post-processing tool that “stitches” the final components of the image together. 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 bootloader/mcuboot/boot/zephyr -d build-mcuboot
west build -b YOUR_BOARD 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 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.

### Extending signing externally

The signing script used when running west flash can be extended or replaced to change features or introduce different signing mechanisms. By default with MCUboot enabled, signing is setup by the cmake/mcuboot.cmake file in Zephyr which adds extra post build commands for generating the signed images. The file used for signing can be replaced from a sysbuild scope (if being used) or from a zephyr/zephyr module scope, the priority of which is:

- Sysbuild
- Zephyr property
- Default MCUboot script (if enabled)

From sysbuild, `-D<target>_SIGNING_SCRIPT` can be used to set a signing script for a specific image or `-DSIGNING_SCRIPT` can be used to set a signing script for all images, for example:

```
west build -b <board> <application> -DSIGNING_SCRIPT=<file>
```

The zephyr property method is achieved by adjusting the SIGNING_SCRIPT property on the zephyr_property_target, ideally from by a module by using:

```c
if(CONFIG_BOOTLOADER_MCUBOOT)
    set_target_properties(zephyr_property_target PROPERTIES SIGNING_SCRIPT ${CMAKE_CURRENT_LIST_DIR}/custom_signing.cmake)
endif()
```

---

2.11. West (Zephyr's meta-tool)
This will include the custom signing CMake file instead of the default Zephyr one when projects are built with MCUboot signing support enabled. The base Zephyr MCUboot signing file can be used as a reference for creating a new signing system or extending the default behaviour.

rimage

rimage configuration uses a different approach that does not rely on Kconfig or CMake but on `west config` instead, similar to Permanent CMake Arguments.

Signing involves a number of “wrapper” scripts stacked on top of each other: `west flash` invokes `west build` which invokes `cmake` and `ninja` which invokes `west sign` which invokes `imgtool` or `rimage`. As long as the signing parameters desired are the default ones and fairly static, these indirections are not a problem. On the other hand, passing `imgtool` or `rimage` options through all these layers can causes issues typical when the layers don't abstract anything. First, this usually requires boilerplate code in each layer. Quoting whitespace or other special characters through all the wrappers can be difficult. Reproducing a lower `west sign` command to debug some build-time issue can be very time-consuming: it requires at least enabling and searching verbose build logs to find which exact options were used. Copying these options from the build logs can be unreliable: it may produce different results because of subtle environment differences. Last and worst: new signing feature and options are impossible to use until more boilerplate code has been added in each layer.

To avoid these issues, `rimage` parameters can bet set in `west config` instead. Here's a workspace/.west/config example:

```plaintext
[sign]
# Not needed when invoked from CMake
tool = rimage

[rimage]
# Quoting is optional and works like in Unix shells
# Not needed when rimage can be found in the default PATH
path = "/home/me/zworkspace/build-rimage/rimage"

# Not needed when using the default development key
extra-args = -i 4 -k 'keys/key argument with space.pem'
```

In order to support quoting, values are parsed by Python's `shlex.split()` like in One-Time CMake Arguments.

The extra-args are passed directly to the `rimage` command. The example above has the same effect as appending them on command line after `--` like this: `west sign --tool rimage -- -i 4 -k 'keys/key argument with space.pem'`. In case both are used, the command-line arguments go last.

2.11.12 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:

```plaintext
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:

```
wes boards -h
```

**Shell completion scripts:** `west completion`

The completion extension command outputs shell completion scripts that can then be used directly to enable shell completion for the supported shells.

It currently supports the following shells:

- bash
- zsh

Additional instructions are available in the command's help:

```
west help completion
```

**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. Enable `CONFIG_BUILD_OUTPUT_META` in your project.

3. Build your application using this pre-created build directory, like so:
4. 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 clause 6](https://spdx.org/licenses/) 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.

**Working with binary blobs:** `west blobs`

The `blobs` command allows users to interact with *binary blobs* declared in one or more *modules* via their `module.yml` file.

The `blobs` command has three sub-commands, used to list, fetch or clean (i.e. delete) the binary blobs themselves.

You can list binary blobs while specifying the format of the output:

```
west blobs list -f '{module}: {type} {path}'
```

For the full set of variables available in `-f/--format` run `west blobs -h`.

Fetching blobs works in a similar manner:

```
wst bikes fetch
```

Note that, as described in the *modules section*, fetched blobs are stored in a `zephyr/blobs/` folder relative to the root of the corresponding module repository.

As does deleting them:
Additionally the tool allows you to specify the modules you want to list, fetch or clean blobs for by typing the module names as a command-line parameter.

**Twister wrapper: west twister**

This command is a wrapper for *twister*.

Twister can then be invoked via west as follows:

```
west twister -help
west twister -T tests/ztest/base
```

**Working with binary descriptors: west bindesc**

The *bindesc* command allows users to read *binary descriptors* of executable files. It currently supports .bin, .hex, .elf and .uf2 files as input.

You can search for a specific descriptor in an image, for example:

```
w west bindesc search KERNEL_VERSION_STRING build/zephyr/zephyr.bin
```

You can search for a custom descriptor by type and ID, for example:

```
w west bindesc custom_search STR 0x200 build/zephyr/zephyr.bin
```

You can dump all of the descriptors in an image using:

```
w west bindesc dump build/zephyr/zephyr.bin
```

You can list all known standard descriptor names using:

```
w west bindesc list
```

## 2.11.13 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.
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 hardcoded 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.

### 2.11.14 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:
   ```bash
   pip3 install --user -U west
   ```
   
   On Windows and macOS:
   ```bash
   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:
   ```bash
   mkdir zephyrproject
   mv zephyr zephyrproject
   cd zephyrproject
   ```
   
   On Windows cmd.exe:
   ```bash
   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:
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.

2.11.15 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.

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 -Bbuild -GNinja -DZEPHYR_MODULES=module1;module2;... samples/hello_world
ninja -Cbuild
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.

Similarly, if your application requires binary blobs and you are not using west, you will need to download and place those blobs in the right places instead of using west blobs. See Binary Blobs 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:

```bash
# 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:

```ini
[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.

**2.12 Testing**

**2.12.1 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.

\(^1\) Note that *west* build invokes *ninja*, among other tools. There’s no recursive invocation of either *west* or *ninja* involved by default, however; *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.
Creating a test suite

Using Ztest to create a test suite is as easy as calling the ZTEST_SUITE. The macro accepts the following arguments:

- **suite_name** - The name of the suite. This name must be unique within a single binary.
- **ztest_suite_predicate_t** - An optional predicate function to allow choosing when the test will run. The predicate will get a pointer to the global state passed in through ztest_run_all() and should return a boolean to decide if the suite should run.
- **ztest_suite_setup_t** - An optional setup function which returns a test fixture. This will be called and run once per test suite run.
- **ztest_suite_before_t** - An optional before function which will run before every single test in this suite.
- **ztest_suite_after_t** - An optional after function which will run after every single test in this suite.
- **ztest_suite_teardown_t** - An optional teardown function which will run at the end of all the tests in the suite.

Below is an example of a test suite using a predicate:

```c
#include <zephyr/ztest.h>
#include "test_state.h"

static bool predicate(const void *global_state)
{
    return ((const struct test_state*)global_state)->x == 5;
}

ZTEST_SUITE(alternating_suite, predicate, NULL, NULL, NULL, NULL);
```

Adding tests to a suite

There are 4 macros used to add a test to a suite, they are:

- **ZTEST** (suite_name, test_name) - Which can be used to add a test by test_name to a given suite by suite_name.
- **ZTEST_USER** (suite_name, test_name) - Which behaves the same as ZTEST, only that when CONFIG_USERSPACE is enabled, then the test will be run in a userspace thread.
- **ZTEST_F** (suite_name, test_name) - Which behaves the same as ZTEST, only that the test function will already include a variable named fixture with the type <suite_name>_fixture.
- **ZTEST_USER_F** (suite_name, test_name) - Which combines the fixture feature of ZTEST_F with the userspace threading for the test.

Test fixtures Test fixtures can be used to help simplify repeated test setup operations. In many cases, tests in the same suite will require some initial setup followed by some form of reset between each test. This is achieved via fixtures in the following way:

```c
#include <zephyr/ztest.h>

struct my_suite_fixture {
    size_t max_size;
    size_t size;
    uint8_t buf[1];
};
```
```c
static void *my_suite_setup(void)
{
    /* Allocate the fixture with 256 byte buffer */
    struct my_suite_fixture *fixture = malloc(sizeof(struct my_suite_fixture) + 255);
    zassume_not_null(fixture, NULL);
    fixture->max_size = 256;
    return fixture;
}

static void my_suite_before(void *f)
{
    struct my_suite_fixture *fixture = (struct my_suite_fixture *)f;
    memset(fixture->buff, 0, fixture->max_size);
    fixture->size = 0;
}

static void my_suite_teardown(void *f)
{
    free(f);
}

ZTEST_SUITE(my_suite, NULL, my_suite_setup, my_suite_before, NULL, my_suite_teardown);

ZTEST_F(my_suite, test_feature_x)
{
    zassert_equal(0, fixture->size);
    zassert_equal(256, fixture->max_size);
}
```

Using memory allocated by a test fixture in a userspace thread, such as during execution of `ZTEST_USER` or `ZTEST_USER_F`, requires that memory to be declared userspace accessible. This is because the fixture memory is owned and initialized by kernel space. The Ztest framework provides the `ZTEST_DMEM` and `ZTEST_BMEM` macros for use of such user/kernel space shared memory.

**Advanced features**

**Test result expectations** Some tests were made to be broken. In cases where the test is expected to fail or skip due to the nature of the code, it’s possible to annotate the test as such. For example:

```c
#include <zephyr/ztest.h>

ZTEST_SUITE(my_suite, NULL, NULL, NULL, NULL, NULL);

ZTEST_EXPECT_FAIL(my_suite, test_fail);
ZTEST(my_suite, test_fail)
{
    /** This will fail the test */
    zassert_true(false, NULL);
}

ZTEST_EXPECT_SKIP(my_suite, test_skip);
ZTEST(my_suite, test_skip)
{
    /** This will skip the test */
}
```

(continues on next page)
In this example, the above tests should be marked as failed and skipped respectively. Instead, Ztest will mark both as passed due to the expectation.

**Test rules**  Test rules are a way to run the same logic for every test and every suite. There are a lot of cases where you might want to reset some state for every test in the binary (regardless of which suite is currently running). As an example, this could be to reset mocks, reset emulators, flush the UART, etc.:

```c
#include <zephyr/fff.h>
#include <zephyr/ztest.h>
#include "test_mocks.h"

DEFINE_FFF_GLOBALS;
DEFINE_FAKE_VOID_FUN(my_weak_func);
static void fff_reset_rule_before(const struct ztest_unit_test *test, void *fixture) {
    ARG_UNUSED(test);
    ARG_UNUSED(fixture);
    RESET_FAKE(my_weak_func);
}

ZTEST_RULE(fff_reset_rule, fff_reset_rule_before, NULL);
```

**A custom test_main**  While the Ztest framework provides a default test_main() function, it's possible that some applications will want to provide custom behavior. This is particularly true if there's some global state that the tests depend on and that state either cannot be replicated or is difficult to replicate without starting the process over. For example, one such state could be a power sequence. Assuming there's a board with several steps in the power-on sequence a test suite can be written using the predicate to control when it would run. In that case, the test_main() function can be written as follows:

```c
#include <zephyr/ztest.h>
#include "my_test.h"

void test_main(void) {
    struct power_sequence_state state;
    /* Only suites that use a predicate checking for phase == PWR_PHASE_0 will run. */
    state.phase = PWR_PHASE_0;
    ztest_run_all(&state, false, 1, 1);
    /* Only suites that use a predicate checking for phase == PWR_PHASE_1 will run. */
    state.phase = PWR_PHASE_1;
    ztest_run_all(&state, false, 1, 1);
    /* Only suites that use a predicate checking for phase == PWR_PHASE_2 will run. */
    state.phase = PWR_PHASE_2;
    ztest_run_all(&state, false, 1, 1);
}
```
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:

```bash
dir/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:

```bash
dir/scripts/twister -T tests/foo/bar/
```

The sample contains the following files:

CMakeLists.txt

```plaintext
# 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

```plaintext
tests:
  # section.subsection
  sample.testing.ztest:
    build_only: true
    platform_allow:
      - native_posix
      - native_sim
    integration_platforms:
      - native_sim
    tags: test_framework
```

prj.conf

```plaintext
CONFIG_ZTEST=y
```

src/main.c (see best practices)

```plaintext
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/ztest.h>
```
Zephyr Project Documentation, Release 3.6.99

(continued from previous page)

```c
ZTEST_SUITE(framework_tests, NULL, NULL, NULL, NULL, NULL);
/**
 * @brief Test Asserts
 *
 * This test verifies various assert macros provided by ztest.
 */
ZTEST(framework_tests, test_assert)
{
    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");
}
```

- **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

For example:

```c
/**
 * @brief Test Asserts
 *
 * This test verifies the zassert_true macro.
 */
ZTEST(my_suite, test_assert)
{
    zassert_true(1, "1 was false");
}
```

**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 running:

```
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()` or `Z_TEST_SKIP_IFDEF`. If the test runs, it needs to report either a pass or fail. For example:

```c
#ifndef CONFIG_TEST1
ZTEST(common, test_test1)
{
    zassert_true(1, "true");
}
#else
ZTEST(common, test_test1)
{
    ztest_test_skip();
}
#endif
ZTEST(common, test_test2)
{
    Z_TEST_SKIP_IFDEF(CONFIG_BUGxxxxx);
    zassert_equal(1, 0, NULL);
}
ZTEST_SUITE(common, NULL, NULL, NULL, NULL, NULL);
```

**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.

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.

Stress test framework

Zephyr stress test framework (Ztress) provides an environment for executing user functions in multiple priority contexts. It can be used to validate that code is resilient to preemptions. The framework tracks the number of executions and preemptions for each context. Execution can have various completion conditions like timeout, number of executions or number of preemptions.

The framework is setting up the environment by creating the requested number of threads (each on different priority), optionally starting a timer. For each context, a user function (different for each context) is called and then the context sleeps for a randomized amount of system ticks. The framework is tracking CPU load and adjusts sleeping periods to achieve higher CPU load. In order to increase the probability of preemptions, the system clock frequency should be relatively high. The default 100 Hz on QEMU x86 is much too low and it is recommended to increase it to 100 kHz.

The stress test environment is setup and executed using `ZTRESS_EXECUTE` which accepts a variable number of arguments. Each argument is a context that is specified by `ZTRESS_TIMER` or `ZTRESS_THREAD` macros. Contexts are specified in priority descending order. Each context specifies completion conditions by providing the minimum number of executions and preemptions. When all conditions are met and the execution has completed, an execution report is printed and the macro returns. Note that while the test is executing, a progress report is periodically printed.

Execution can be prematurely completed by specifying a test timeout (`ztress_set_timeout()`) or an explicit abort (`ztress_abort()`).

User function parameters contains an execution counter and a flag indicating if it is the last execution.

The example below presents how to setup and run 3 contexts (one of which is k_timer interrupt handler context). Completion criteria is set to at least 10000 executions of each context and 1000 preemptions of the lowest priority context. Additionally, the timeout is configured to complete after 10 seconds if those conditions are not met. The last argument of each context is the initial sleep time which will be adjusted throughout the test to achieve the highest CPU load.

```
ztress_set_timeout(K_MSEC(10000));
ZTRESS_EXECUTE(ZTRESS_TIMER(foo_0, user_data_0, 10000, Z_TIMEOUT_TICKS(20)),
               ZTRESS_THREAD(foo_1, user_data_1, 10000, 0, Z_TIMEOUT_TICKS(20)),
               ZTRESS_THREAD(foo_2, user_data_2, 10000, 1000, Z_TIMEOUT_TICKS(20)));
```

Configuration  Static configuration of Ztress contains:

• `ZTRESS_MAX_THREADS` - number of supported threads.
• `ZTRESS_STACK_SIZE` - Stack size of created threads.
• `ZTRESS_REPORT_PROGRESS_MS` - Test progress report interval.

API reference

Running tests
This module eases the testing process by providing helpful macros and other testing structures.

Defines

ZTEST(suite, fn)

Create and register a new unit test.

Calling this macro will create a new unit test and attach it to the declared suite. The suite does not need to be defined in the same compilation unit.

Parameters

- suite – The name of the test suite to attach this test
- fn – The test function to call.

ZTEST_USER(suite, fn)

Define a test function that should run as a user thread.

This macro behaves exactly the same as ZTEST, but calls the test function in user space if CONFIG_USERSPACE was enabled.

Parameters

- suite – The name of the test suite to attach this test
- fn – The test function to call.

ZTEST_F(suite, fn)

Define a test function.

This macro behaves exactly the same as ZTEST(), but the function takes an argument for the fixture of type struct suite##_fixture* named fixture.

Parameters

- suite – The name of the test suite to attach this test
- fn – The test function to call.

ZTEST_USER_F(suite, fn)

Define a test function that should run as a user thread.

If CONFIG_USERSPACE is not enabled, this is functionally identical to ZTEST_F(). The test function takes a single fixture argument of type struct suite##_fixture* named fixture.

Parameters

- suite – The name of the test suite to attach this test
- fn – The test function to call.

ZTEST_RULE(name, before_each_fn, after_each_fn)

Define a test rule that will run before/after each unit test.

Functions defined here will run before/after each unit test for every test suite. Along with the callback, the test functions are provided a pointer to the test being run, and the data. This provides a mechanism for tests to perform custom operations depending on the specific test or the data (for example logging may use the test's name).

Ordering:
• Test rule's before function will run before the suite's before function. This is done to allow the test suite's customization to take precedence over the rule which is applied to all suites.

• Test rule's after function is not guaranteed to run in any particular order.

**Parameters**

- **name** – The name for the test rule (must be unique within the compilation unit)
- **before_each_fn** – The callback function (ztest_rule_cb) to call before each test (may be NULL)
- **after_each_fn** – The callback function (ztest_rule_cb) to call after each test (may be NULL)

```c
ztest_run_test_suite(suite, shuffle, suite_iter, case_iter)
```
Run the specified test suite.

**Parameters**

- **suite** – Test suite to run.
- **shuffle** – Shuffle tests
- **suite_iter** – Test suite repetitions.
- **case_iter** – Test case repetitions.

**Typedefs**

```c
typedef void (*ztest_rule_cb)(const struct ztest_unit_test *test, void *data)
```
Test rule callback function signature.

The function signature that can be used to register a test rule's before/after callback. This provides access to the test and the fixture data (if provided).

- **Param test**
  Pointer to the unit test in context

- **Param data**
  Pointer to the test's fixture data (may be NULL)

**Functions**

```c
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.
```

```c
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.
```

```c
void ztest_test_skip(void)
Skip the current test.
```
void ztest_skip_failed_assumption(void)

void ztest_simple_1cpu_before(void *data)
   A 'before' function to use in test suites that just need to start 1cpu.
   Ignores data, and calls z_test_1cpu_start()

   Parameters
   • data – The test suite's data

void ztest_simple_1cpu_after(void *data)
   A 'after' function to use in test suites that just need to stop 1cpu.
   Ignores data, and calls z_test_1cpu_stop()

   Parameters
   • data – The test suite's data

struct ztest_test_rule

struct ztest_arch_api
   #include <ztest_test.h> Structure for architecture specific APIs.

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 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, ...)
zassume(cond, default_msg, ...)
zexpect(cond, default_msg, ...)
zassert_unreachable(...)
   Assert that this function call won’t be reached.

   Parameters
   • ... – Optional message and variables to print if the assertion fails

zassert_true(cond, ...)
   Assert that cond is true.

   Parameters
   • cond – Condition to check
   • ... – Optional message and variables to print if the assertion fails
zassert_false(cond, ...)  
Assert that cond is false.

Parameters
- cond – Condition to check
- ... – Optional message and variables to print if the assertion fails

zassert_ok(cond, ...)  
Assert that cond is 0 (success)

Parameters
- cond – Condition to check
- ... – Optional message and variables to print if the assertion fails

zassert_not_ok(cond, ...)  
Assert that cond is not 0 (failure)

Parameters
- cond – Condition to check
- ... – Optional message and variables to print if the assertion fails

zassert_is_null(ptr, ...)  
Assert that ptr is NULL.

Parameters
- ptr – Pointer to compare
- ... – Optional message and variables to print if the assertion fails

zassert_not_null(ptr, ...)  
Assert that ptr is not NULL.

Parameters
- ptr – Pointer to compare
- ... – Optional message and variables to print if the assertion fails

zassert_equal(a, b, ...)  
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
- ... – Optional message and variables to print if the assertion fails

zassert_not_equal(a, b, ...)  
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
- ... – Optional message and variables to print if the assertion fails
zassert_equal_ptr(a, b, ...)
Assert that \( a \) equals \( b \).
\( a \) and \( b \) will be converted to void * before comparing.

**Parameters**
- \( a \) – Value to compare
- \( b \) – Value to compare
- \( ... \) – Optional message and variables to print if the assertion fails

zassert_within(a, b, d, ...)
Assert that \( a \) is within \( b \) with delta \( d \).

**Parameters**
- \( a \) – Value to compare
- \( b \) – Value to compare
- \( d \) – Delta
- \( ... \) – Optional message and variables to print if the assertion fails

zassert_between_inclusive(a, l, u, ...)
Assert that \( a \) is greater than or equal to \( l \) and less than or equal to \( u \).

**Parameters**
- \( a \) – Value to compare
- \( l \) – Lower limit
- \( u \) – Upper limit
- \( ... \) – Optional message and variables 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, ...)
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
- \( ... \) – Optional message and variables to print if the assertion fails
**Expectations**  These macros will continue test execution if the related expectation fails and subsequently fail the test at the end of its execution. When an expectation fails, it will print the current file, line, and function, alongside a reason for the failure and an optional message but continue executing the test. If the config `CONFIG_ZTEST_ASSERT_VERBOSE` is 0, the expectations will only print the file and line numbers, reducing the binary size of the test.

For example, if the following expectations fail:

```c
zexpect_equal(buf->ref, 2, "Invalid refcount");
zexpect_equal(buf->ref, 1337, "Invalid refcount");
```

The output will look something like:

```
START - test_get_single_buffer
    Expectation failed at main.c:62: test_get_single_buffer: Invalid refcount (buf->ref not equal to 2)
    Expectation failed at main.c:63: test_get_single_buffer: Invalid refcount (buf->ref not equal to 1337)
FAIL - test_get_single_buffer in 0.0 seconds
```

**group ztest_expect**

This module provides expectations when using Ztest.

**Defines**

**zexpect_true**(cond, ...)

Expect that `cond` is true, otherwise mark test as failed but continue its execution.

**Parameters**

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails

**zexpect_false**(cond, ...)

Expect that `cond` is false, otherwise mark test as failed but continue its execution.

**Parameters**

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails

**zexpect_ok**(cond, ...)

Expect that `cond` is 0 (success), otherwise mark test as failed but continue its execution.

**Parameters**

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails

**zexpect_not_ok**(cond, ...)

Expect that `cond` is not 0 (failure), otherwise mark test as failed but continue its execution.

**Parameters**

- `cond` – Condition to check
- `...` – Optional message and variables to print if the expectation fails
zexpect_is_null(ptr, ...)
   Expect that `ptr` is NULL, otherwise mark test as failed but continue its execution.

   Parameters
   • `ptr` – Pointer to compare
   • `...` – Optional message and variables to print if the expectation fails

zexpect_not_null(ptr, ...)
   Expect that `ptr` is not NULL, otherwise mark test as failed but continue its execution.

   Parameters
   • `ptr` – Pointer to compare
   • `...` – Optional message and variables to print if the expectation fails

zexpect_equal(a, b, ...)
   Expect that `a` equals `b`, otherwise mark test as failed but continue its execution.

   Parameters
   • `a` – Value to compare
   • `b` – Value to compare
   • `...` – Optional message and variables to print if the expectation fails

zexpect_not_equal(a, b, ...)
   Expect that `a` does not equal `b`, otherwise mark test as failed but continue its execution.

   `a` and `b` won’t be converted and will be compared directly.

   Parameters
   • `a` – Value to compare
   • `b` – Value to compare
   • `...` – Optional message and variables to print if the expectation fails

zexpect_equal_ptr(a, b, ...)
   Expect that `a` equals `b`, otherwise mark test as failed but continue its execution.

   `a` and `b` will be converted to `void *` before comparing.

   Parameters
   • `a` – Value to compare
   • `b` – Value to compare
   • `...` – Optional message and variables to print if the expectation fails

zexpect_within(a, b, delta, ...)
   Expect that `a` is within `b` with delta `d`, otherwise mark test as failed but continue its execution.

   Parameters
   • `a` – Value to compare
   • `b` – Value to compare
   • `delta` – Difference between `a` and `b`
   • `...` – Optional message and variables to print if the expectation fails
Zephyr Project Documentation, Release 3.6.99

zexpect_between_inclusive(a, lower, upper, ...)

Expect that $a$ is greater than or equal to $l$ and less than or equal to $u$, otherwise mark test as failed but continue its execution.

**Parameters**

- $a$ – Value to compare
- $lower$ – Lower limit
- $upper$ – Upper limit
- ... – Optional message and variables to print if the expectation fails

zexpect_mem_equal(buf, exp, size, ...)

Expect that 2 memory buffers have the same contents, otherwise mark test as failed but continue its execution.

**Parameters**

- $buf$ – Buffer to compare
- $exp$ – Buffer with expected contents
- $size$ – Size of buffers
- ... – Optional message and variables to print if the expectation fails

**Assumptions**

These macros will instantly skip the test or suite if the related assumption fails. When an assumption fails, it will print the current file, line, and function, alongside a reason for the failure and an optional message. If the config CONFIG_ZTEST_ASSERT_VERBOSE is 0, the assumptions will only print the file and line numbers, reducing the binary size of the test.

Example output for a failed macro from zassume_equal(buf->ref, 2, "Invalid refcount"):

group ztest_assume

This module provides assumptions when using Ztest.

**Defines**

zassume_true(cond, ...)

Assume that $cond$ is true.

If the assumption fails, the test will be marked as “skipped”.

**Parameters**

- $cond$ – Condition to check
- ... – Optional message and variables to print if the assumption fails

zassume_false(cond, ...)

Assume that $cond$ is false.

If the assumption fails, the test will be marked as “skipped”.

**Parameters**

- $cond$ – Condition to check
- ... – Optional message and variables to print if the assumption fails
zassume_ok(cond, ...)  
Assume that \texttt{cond} is 0 (success)  
If the assumption fails, the test will be marked as “skipped”.  

\textbf{Parameters}  
- \texttt{cond} – Condition to check  
- \ldots – Optional message and variables to print if the assumption fails

zassume_not_ok(cond, ...)  
Assume that \texttt{cond} is not 0 (failure)  
If the assumption fails, the test will be marked as “skipped”.  

\textbf{Parameters}  
- \texttt{cond} – Condition to check  
- \ldots – Optional message and variables to print if the assumption fails

zassume_is_null(ptr, ...)  
Assume that \texttt{ptr} is NULL.  
If the assumption fails, the test will be marked as “skipped”.  

\textbf{Parameters}  
- \texttt{ptr} – Pointer to compare  
- \ldots – Optional message and variables to print if the assumption fails

zassume_not_null(ptr, ...)  
Assume that \texttt{ptr} is not NULL.  
If the assumption fails, the test will be marked as “skipped”.  

\textbf{Parameters}  
- \texttt{ptr} – Pointer to compare  
- \ldots – Optional message and variables to print if the assumption fails

zassume_equal(a, b, ...)  
Assume that \texttt{a} equals \texttt{b}.  
\texttt{a} and \texttt{b} won’t be converted and will be compared directly. If the assumption fails, the test will be marked as “skipped”.  

\textbf{Parameters}  
- \texttt{a} – Value to compare  
- \texttt{b} – Value to compare  
- \ldots – Optional message and variables to print if the assumption fails

zassume_not_equal(a, b, ...)  
Assume that \texttt{a} does not equal \texttt{b}.  
\texttt{a} and \texttt{b} won’t be converted and will be compared directly. If the assumption fails, the test will be marked as “skipped”.  

\textbf{Parameters}  
- \texttt{a} – Value to compare  
- \texttt{b} – Value to compare  
- \ldots – Optional message and variables to print if the assumption fails
zassume_equal_ptr(a, b, ...)
Assume that \( a \) equals \( b \).
\( a \) and \( b \) will be converted to `void *` before comparing. If the assumption fails, the test will be marked as “skipped”.

Parameters
- \( a \) – Value to compare
- \( b \) – Value to compare
- \( ... \) – Optional message and variables to print if the assumption fails

zassume_within(a, b, d, ...)
Assume that \( a \) is within \( b \) with delta \( d \).
If the assumption fails, the test will be marked as “skipped”.

Parameters
- \( a \) – Value to compare
- \( b \) – Value to compare
- \( d \) – Delta
- \( ... \) – Optional message and variables to print if the assumption fails

zassume_between_inclusive(a, l, u, ...)
Assume that \( a \) is greater than or equal to \( l \) and less than or equal to \( u \).
If the assumption fails, the test will be marked as “skipped”.

Parameters
- \( a \) – Value to compare
- \( l \) – Lower limit
- \( u \) – Upper limit
- \( ... \) – Optional message and variables to print if the assumption fails

zassume_mem_equal(...)
Assume that 2 memory buffers have the same contents.

This macro calls the final memory comparison assumption 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 `zassume_mem_equal__` for real arguments accepted.

zassume_mem_equal__ (buf, exp, size, ...)
Internal assume that 2 memory buffers have the same contents.
If the assumption fails, the test will be marked as “skipped”.

Note: This is internal macro, to be used as a second expansion. See `zassume_mem_equal`.

Parameters
- \( buf \) – Buffer to compare
- \( exp \) – Buffer with expected contents
• size – Size of buffers
• ... – Optional message and variables to print if the assumption fails

Ztress

**group ztest_ztress**

This module provides test stress when using Ztest.

**Defines**

**ZTRESS_TIMER**(handler, user_data, exec_cnt, init_timeout)
Descriptor of a k_timer handler execution context.

The handler is executed in the k_timer handler context which typically means interrupt context. This context will preempt any other used in the set.

**Note:** There can only be up to one k_timer context in the set and it must be the first argument of **ZTRESS_EXECUTE**.

**Parameters**

• **handler** – User handler of type *ztress_handler*.
• **user_data** – User data passed to the handler.
• **exec_cnt** – Number of handler executions to complete the test. If 0 then this is not included in completion criteria.
• **init_timeout** – Initial backoff time base (given in *k_timeout_t*). It is adjusted during the test to optimize CPU load. The actual timeout used for sleeping is randomized.

**ZTRESS_THREAD**(handler, user_data, exec_cnt, preempt_cnt, init_timeout)
Descriptor of a thread execution context.

The handler is executed in the thread context. The priority of the thread is determined based on the order in which contexts are listed in **ZTRESS_EXECUTE**.

**Note:** thread sleeps for random amount of time. Additionally, the thread busy-waits for a random length of time to further increase randomization in the test.

**Parameters**

• **handler** – User handler of type *ztress_handler*.
• **user_data** – User data passed to the handler.
• **exec_cnt** – Number of handler executions to complete the test. If 0 then this is not included in completion criteria.
• **preempt_cnt** – Number of preemptions of that context to complete the test. If 0 then this is not included in completion criteria.
• **init_timeout** – Initial backoff time base (given in *k_timeout_t*). It is adjusted during the test to optimize CPU load. The actual timeout used for sleeping is randomized.
ZTRESS_CONTEXT_INITIALIZER(_handler, _user_data, _exec_cnt, _preempt_cnt, _t)
Initialize context structure.

For argument types see ztress_context_data. For more details see ZTRESS_THREAD.

Parameters
- _handler – Handler.
- _user_data – User data passed to the handler.
- _exec_cnt – Execution count limit.
- _preempt_cnt – Preemption count limit.
- _t – Initial timeout.

ZTRESS_EXECUTE(...)  
Setup and run stress test.

It initialises all contexts and calls ztress_execute.

Parameters
- ... – List of contexts. Contexts are configured using ZTRESS_TIMER and ZTRESS_THREAD macros. ZTRESS_TIMER must be the first argument if used. Each thread context has an assigned priority. The priority is assigned in a descending order (first listed thread context has the highest priority). The maximum number of supported thread contexts, including the timer context, is configurable in Kconfig (ZTRESS_MAX_THREADS).

Typedefs

typedef bool (*ztress_handler)(void *user_data, uint32_t cnt, bool last, int prio)
User handler called in one of the configured contexts.

Param user_data
User data provided in the context descriptor.

Param cnt
Current execution counter. Counted from 0.

Param last
Flag set to true indicates that it is the last execution because completion criteria are met, test timed out or was aborted.

Param prio
Context priority counting from 0 which indicates the highest priority.

Retval true
continue test.

Retval false
stop executing the current context.

Functions

int ztress_execute(struct ztress_context_data *timer_data, struct ztress_context_data *thread_data, size_t cnt)
Execute contexts.

The test runs until all completion requirements are met or until the test times out (use ztress_set_timeout to configure timeout) or until the test is aborted (ztress_abort).
on test completion a report is printed (ztress_report is called internally).

**Parameters**

- timer_data – Timer context. NULL if timer context is not used.
- thread_data – List of thread contexts descriptors in priority descending order.
- cnt – Number of thread contexts.

**Return values**

- EINVAL – If configuration is invalid.
- 0 – if test is successfully performed.

```c
void ztress_abort(void)
Abort ongoing stress test.
```

```c
void ztress_set_timeout(k_timeout_t t)
Set test timeout.
Test is terminated after timeout disregarding completion criteria. Setting is persistent between executions.
```

**Parameters**

- t – Timeout.

```c
void ztress_report(void)
Print last test report.
Report contains number of executions and preemptions for each context, initial and adjusted timeouts and CPU load during the test.
```

```c
int ztress_exec_count(uint32_t id)
Get number of executions of a given context in the last test.
```

**Parameters**

- id – Context id. 0 means the highest priority.

**Returns**

Number of executions.

```c
int ztress_preempt_count(uint32_t id)
Get number of preemptions of a given context in the last test.
```

**Parameters**

- id – Context id. 0 means the highest priority.

**Returns**

Number of preemptions.

```c
uint32_t ztress_optimized_ticks(uint32_t id)
Get optimized timeout base of a given context in the last test.
Optimized value can be used to update initial value. It will improve the test since optimal CPU load will be reach immediately.
```

**Parameters**

- id – Context id. 0 means the highest priority.

**Returns**

Optimized timeout base.
struct ztress_context_data
   
#include <ztress.h>

Mocking via FFF  Zephyr has integrated with FFF for mocking. See FFF for documentation. To use it, include the relevant header:

#include <zephyr/fff.h>

Zephyr provides several FFF-based fake drivers which can be used as either stubs or mocks. Fake driver instances are configured via Devicetree and Configuration System (Kconfig). See the following devicetree bindings for more information:

- zephyr,fake-can
- zephyr,fake-eeprom

Zephyr also has defined extensions to FFF for simplified declarations of fake functions. See FFF Extensions.

Customizing Test 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/zephyr/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 */
```

Shuffling Test Sequence

By default the tests are sorted and ran in alphanumerical order. Test cases may be dependent on this sequence. Enable CONFIG_ZTEST_SHUFFLE to randomize the order. The output from the test will display the seed for failed tests. For native simulator builds you can provide the seed as an argument to twister with –seed

Static configuration of ZTEST_SHUFFLE contains:

- CONFIG_ZTEST_SHUFFLE_SUITE_REPEAT_COUNT - Number of iterations the test suite will run.
- CONFIG_ZTEST_SHUFFLE_TEST_REPEAT_COUNT - Number of iterations the test will run.

Test Selection

For tests built for native simulator, use command line arguments to list or select tests to run. The test argument expects a comma separated list of suite::test. You can substitute the test name with an * to run all tests within a suite.

For example
FFF Extensions

group fff_extensions

This module provides extensions to FFF for simplifying the configuration and usage of fakes.

Defines

RETURN_HANDLED_CONTEXT(FUNCNAME, CONTEXTTYPE, RESULTFIELD,
CONTEXTPTRNAME, HANDLERBODY)

Wrap custom fake body to extract defined context struct.

Add extension macro for simplified creation of fake functions needing call-specific context data.

This macro enables a fake to be implemented as follows and requires no familiarity with the inner workings of FFF.

```c
struct FUNCNAME##_custom_fake_context
{
    struct instance * const instance;
    int result;
};

int FUNCNAME##_custom_fake(
    const struct instance **instance_out)
{
    RETURN_HANDLED_CONTEXT(
        FUNCNAME,
        struct FUNCNAME##_custom_fake_context,
        result,
        context,
        {
            if (context != NULL)
            {
                if (context->result == 0)
                {
                    if (instance_out != NULL)
                    {
                        *instance_out = context->instance;
                    }
                }
                return context->result;
            }
            return FUNCNAME##_fake.return_val;
        }
    );
}
```

**Parameters**

- FUNCNAME – Name of function being faked
2.12.2 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_sim, 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.

To run the script in the local tree, follow the steps below:

Linux

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

Windows

```
zephyr-env.cmd
python ./scripts\twister
```

If you have a system with a large number of cores and plenty of free storage space, you can build and run all possible tests using the following options:

Linux

```
$ ./scripts/twister --all --enable-slow
```

Windows

```
python ./scripts\twister --all --enable-slow
```

This will build for all available boards and run all applicable tests in a simulated (for example QEMU) environment.

If you want to run tests on one or more specific platforms, you can use the --platform option, it is a platform filter for testing, with this option, test suites will only be built/run on the platforms specified. This option also supports different revisions of one same board, you can use --platform board@revision to test on a specific revision.
The list of command line options supported by twister can be viewed using:

Linux

```bash
$ ./scripts/twister --help
```

Windows

```python
python .\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:

```yaml
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`:

```bash
# 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

**simulation:**
Simulator used to simulate the platform, e.g. 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.

timeout_multiplier: <float> (default 1)
   Multiply each test case timeout by specified ratio. This option allows to tune timeouts
   only for required platform. It can be useful in case naturally slow platform I.e.: HW
   board with power-efficient but slow CPU or simulation platform which can perform
   instruction accurate simulation but does it slowly.

test:
   A list of environment variables. Twister will check if all these environment variables are
   set, and otherwise skip this platform. This allows the user to define a platform which should
   be used, for example, only if some required software or hardware is present, and to signal
   that presence to twister using these environment variables.

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: alphanumeric 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.semaphore.
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 testing 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
    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 any-
thing. 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 or --enable-slow-only 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
```

Using namespaces, it is possible to apply a configuration only to some hardware. Currently
both architectures and platforms are supported:

```
common:
  tags: drivers adc
tests:
  test:
    depends_on: adc
test_async:
      extra_configs:
        - arch:x86:CONFIG_ADC_ASYNC=y
        - platform:qemu_x86:CONFIG_DEBUG=y
```

build_only: <True|False> (default False)
If true, twister will not try to run the test even if the test is runnable on the platform.

This keyword is reserved for tests that are used to test if some code actually builds. A
build_only test is not designed to be run in any environment and should not be testing
any functionality, it only verifies that the code builds.

This option is often used to test drivers and the fact that they are correctly enabled in Zephyr
and that the code builds, for example sensor drivers. Such test shall not be used to verify
the functionality of the driver.

build_on_all: <True|False> (default False)
If true, attempt to build test on all available platforms. This is mostly used in CI for increased
coverage. Do not use this flag in new tests.

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.

levels: <list of levels>
Test levels this test should be part of. If a level is present, this test will be selectable using
the command line option --level <level name>
**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 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.

**sysbuild**: <True|False> (default False)
Build the project using sysbuild infrastructure. Only the main project's generated device-
tree and Kconfig will be used for filtering tests. On device testing must use the hardware
map, or west flash to load the images onto the target. The --erase option of west flash is
not supported with this option. Usage of unsupported options will result in tests requiring
sysbuild support being skipped.

**harness**: <string>
A harness keyword in the test case.yaml file identifies a Twister harness needed to run
tests successfully. A harness is a feature of Twister and implemented by Twister; some
harnesses are defined as placeholders and have no implementation yet.

A harness can be seen as the handler that needs to be implemented in Twister to be able
to evaluate if a test passes criteria. For example, a keyboard harness is set on tests that re-
quire keyboard interaction to reach verdict on whether a test has passed or failed, however,
Twister lack this harness implementation at the moment.

Supported harnesses:

- ztest
- test
- console
- pytest
- gtest
- robot

Harnesses ztest, gtest and console are based on parsing of the output and matching cer-
tain phrases. ztest and gtest harnesses look for pass/fail/etc. frames defined in those
frameworks. Use gtest harness if you've already got tests written in the gTest framework.
and do not wish to update them to zTest. The console harness tells Twister to parse a test’s text output for a regex defined in the test’s YAML file. The robot harness is used to execute Robot Framework test suites in the Renode simulation framework.

Some widely used harnesses that are not supported yet:

- keyboard
- net
- bluetooth

Harness bsim is implemented in limited way - it helps only to copy the final executable (zephyr.exe) from build directory to BabbleSim’s bin directory (${BSIM_OUT_PATH}/bin). This action is useful to allow BabbleSim’s tests to directly run after. By default, the executable file name is (with dots and slashes replaced by underscores): bs_<platform_name>_<test_path>_<test_scenario_name>. This name can be overridden with the bsim_exe_name option in harness_config section.

**platform_key: [list of platform attributes]**

Often a test needs to only be built and run once to qualify as passing. Imagine a library of code that depends on the platform architecture where passing the test on a single platform for each arch is enough to qualify the tests and code as passing. The platform_key attribute enables doing just that.

For example to key on (arch, simulation) to ensure a test is run once per arch and simulation (as would be most common):

```plaintext
platform_key:
  - arch
  - simulation
```

Adding platform (board) attributes to include things such as soc name, soc family, and perhaps sets of IP blocks implementing each peripheral interface would enable other interesting uses. For example, this could enable building and running SPI tests once for each unique IP block.

**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

- **regex: [list of regular expressions]** (required)
  - Strings with regular expressions to match with the test’s output to confirm the 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

- **record: [recording options]** (optional)
  - **regex: [regular expression]** (required)
    - The regular expression with named subgroups to match data fields at the test’s output lines where the test provides some custom data for further analysis. These records will be written into the build directory ‘recording.csv’ file as well as ‘recording’ property of the test suite object in ‘twister.json’.

    For example, to extract three data fields ‘metric’, ‘cycles’, ‘nanoseconds’:
**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_his221`, `i2c_bme280`, `i2c_FRAM`, `ble_fw` and `gpio_loop`.

Only one fixture can be defined per testcase and the fixture name has to be unique across all tests in the test suite.

**pytest_root**: `<list of pytest testpaths>` (default `pytest`)
Specify a list of pytest directories, files or subtests that need to be executed when a test case begins to run. The default pytest directory is `pytest`. After the pytest run is finished, Twister will check if the test case passed or failed according to the pytest report. As an example, a list of valid pytest roots is presented below:

```yaml
harness_config:
  pytest_root:
    - "pytest/test_shell_help.py"
    - "../shell/pytest/test_shell.py"
    - "~/tmp/test_shell.py"
    - "$ZEPHYR_BASE/samples/subsys/testsuite/pytest/shell/pytest/test_shell.py"
    - "pytest/test_shell_help.py::test_shell2_sample"  # select pytest
    - "pytest/test_shell_help.py::test_shell2_sample[param_a]"  # select
    - "pytest parametrized subtest"
```

**pytest_args**: `<list of arguments>` (default empty)
Specify a list of additional arguments to pass to pytest e.g.: `pytest_args: ['-k=test_method', '--log-level=DEBUG']`. Note that `--pytest-args` can be passed multiple times to pass several arguments to the pytest.

**pytest_dut_scope**: `<function|class|module|package|session>` (default `function`)
The scope for which dut and shell pytest fixtures are shared. If the scope is set to function, DUT is launched for every test case in python script. For `session` scope, DUT is launched only once.

**robot_test_path**: `<robot file path>` (default empty)
Specify a path to a file containing a Robot Framework test suite to be run.

**bsim_exe_name**: `<string>`
If provided, the executable filename when copying to BabbleSim’s bin directory, will be `bs_<platform_name>_\_<bsim_exe_name>` instead of the default based on the test path and scenario name.

The following is an example yaml file with a few harness_config options.
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 examples in samples/subsys testsuite/pytest/.

```yaml
common:
    harness: pytest
tests:
    pytest.example.directories:
        harness_config:
            pytest_root:
                - pytest_dir1
                - $ENV_VAR/samples/test/pytest_dir2
    pytest.example.files_and_subtests:
        harness_config:
            pytest_root:
                - pytest/test_file_1.py
                - test_file_2.py::test_A
                - test_file_2.py::test_B[param_a]
```

The following is an example yaml file with robot harness_config options.

```yaml
tests:
    robot.example:
        harness: robot
        harness_config:
            robot_test_path: [robot file path]
```

**filter: <expression>**

Filter whether the testcase should be run by evaluating an expression against an environment containing the following values:

```yaml
{ ARCH : <architecture>,
  PLATFORM : <platform>,
  <all CONFIG_* key/value pairs in the test's generated defconfig>,
  *<env>: any environment variable available
}
```

Twister will first evaluate the expression to find if a “limited” cmake call, i.e. using package_helper cmake script, can be done. Existence of “dt_*” entries indicates devicetree is needed. Existence of “CONFIG*” entries indicates kconfig is needed. If there are no other types of entries in the expression a filtration can be done without creating a complete build system. If there are entries of other types a full cmake is required.

The grammar for the expression language is as follows:

```yaml
expression : expression 'and' expression
             expression 'or' expression
             'not' expression
             '(' expression ')'
             symbol '==' constant
             symbol '!=' constant
             symbol '<' NUMBER
             symbol '>' NUMBER
             symbol '>=' NUMBER
```
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.

**required_snippets: <list of needed snippets>**

Snippets are supported in twister for test cases that require them. As with normal applications, twister supports using the base zephyr snippet directory and test application directory for finding snippets. Listed snippets will filter supported tests for boards (snippets must be compatible with a board for the test to run on them, they are not optional).

The following is an example yaml file with 2 required snippets.

| tests: |

| snippet.example: |

| required_snippets: |

| - cdc-acm-console |

| - user-snippet-example |

The set of test cases that actually run depends on directives in the testcase field 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, add + 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.

**Managing tests timeouts**

There are several parameters which control tests timeouts on various levels:

- **timeout** option in each test case. See [here](#) for more details.
- **timeout_multiplier** option in board configuration. See [here](#) for more details.
- **--timeout-multiplier** twister option which can be used to adjust timeouts in exact twister run. It can be useful in case of simulation platform as simulation time may depend on the host speed & load or we may select different simulation method (i.e. cycle accurate but slower one), etc...

Overall test case timeout is a multiplication of these three parameters.

**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 custom emulator**

Apart from the already supported QEMU and other simulated environments, Twister supports running any out-of-tree custom emulator defined in the board's `board.cmake`. To use this type of simulation, add the following properties to `custom_board/custom_board.yaml`:

```
simulation: custom
simulation_exec: <name_of_emu_binary>
```

This tells Twister that the board is using a custom emulator called `<name_of_emu_binary>`, make sure this binary exists in the PATH.

Then, in `custom_board/board.cmake`, set the supported emulation platforms to `custom`:

```
set(SUPPORTED_EMU_PLATFORMS custom)
```

Finally, implement the `run_custom` target in `custom_board/board.cmake`. It should look something like this:

```
add_custom_target(run_custom
    COMMAND
    <name_of_emu_binary to invoke during 'run'>
    <any args to be passed to the command, i.e. $(BOARD), $(APPLICATION_BINARY_DIR)/zephyr/...
    WORKING_DIRECTORY $(APPLICATION_BINARY_DIR)
    DEPENDS ${logical_target_for_zephyr_elf}
    USES_TERMINAL
)
```

**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:

Linux

```
scripts/twister --device-testing --device-serial /dev/ttyACM0 \
--device-serial-baud 115200 -p frdm_k64f -T tests/kernel
```

Windows

```
python .\scripts\twister --device-testing --device-serial COM1 \
--device-serial-baud 115200 -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.

The --device-serial-baud option is only needed if your device does not run at 115200 baud.

To support devices without a physical serial port, use the --device-serial-pty option. In this cases, log messages are captured for example using a script. In this case you can run twister with the following options:

Linux

```
scripts/twister --device-testing --device-serial-pty "$script.py" \
-p intel_adsp/cavs25 -T tests/kernel
```

Windows

Note: Not supported on Windows OS

The script is user-defined and handles delivering the messages which can be used by twister to determine the test execution status.

The --device-flash-timeout option allows to set explicit timeout on the device flash operation, for example when device flashing takes significantly large time.

The --device-flash-with-test option indicates that on the platform the flash operation also executes a test case, so the flash timeout is increased by a test case timeout.

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, baud and their IDs if available. Run the following command to produce the hardware map:

Linux

```
./scripts/twister --generate-hardware-map map.yml
```

Windows

```
python .\scripts\twister --generate-hardware-map map.yml
```

The generated hardware map file (map.yml) will have the list of connected devices, for example:

Linux

```
- connected: true
  id: OSHW00032254e4500128002ab98002784d1000097969900
  platform: unknown
  product: DAPLink CMSIS-DAP
```

(continues on next page)
runner: pyocd
serial: /dev/cu.usbmodem146114202
- connected: true
  id: 00683759358
  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 the platform names, the products 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:

Linux

- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: reel_board
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
  baud: 9600
- connected: true
  id: 00683759358
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: nrfjprog
  serial: /dev/cu.usbmodem0006837593581
  baud: 9600

Windows

- connected: true
  id: OSHW000032254e4500128002ab98002784d1000097969900
  platform: reel_board
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
  baud: 9600
- connected: true
  id: 00683759358
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: nrfjprog
  serial: /dev/cu.usbmodem0006837593581
  baud: 9600
The baud entry is only needed if not running at 115200.

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

Linux

```bash
./scripts/twister --device-testing --hardware-map map.yml -T samples/hello_world/
```

Windows

```bash
python .\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 pyocd, nrfjprog, jlink, openocd, or dediprog are supported with the hardware map features. Boards that require other runners to flash the Zephyr binary are still work in progress.

Hardware map allows to set `--device-flash-timeout` and `--device-flash-with-test` command line options as `flash-timeout` and `flash-with-test` fields respectively. These hardware map values override command line options for the particular platform.

Serial PTY support using `--device-serial-pty` can also be used in the hardware map:

```json
- connected: true
  id: None
  platform: intel_adsp/cavs25
  product: None
  runner: intel_adsp
  serial_pty: path/to/script.py
  runner_params:
    - --remote-host=remote_host_ip_addr
    - --key=/path/to/key.pem
```

The runner_params field indicates the parameters you want to pass to the west runner. For some boards the west runner needs some extra parameters to work. It is equivalent to following west and twister commands.

Linux

```bash
west flash --remote-host remote_host_ip_addr --key /path/to/key.pem
twister -p intel_adsp/cavs25 --device-testing --device-serial-pty script.py --west-flash="--remote-host=remote_host_ip_addr,--key=/path/to/key.pem"
```

Windows

**Note:** Not supported on Windows OS

**Note:** For serial PTY, the “--generate-hardware-map” option cannot scan it out and generate a correct hardware map automatically. You have to edit it manually according to above example. This is because the serial port of the PTY is not fixed and being allocated in the system at runtime.
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:

```yaml
- connected: true
  fixtures:
    - gpio_loopback
      id: 0240000026334e450015400f5e0e00b4eb1000097969900
      platform: frdm_k64f
      product: DAPLink CMSIS-DAP
      runner: pyocd
      serial: /dev/ttyACM9
```

When running 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.

Fixtures can also be provided via twister command option `--fixture`, this option can be used multiple times and all given fixtures will be appended as a list. And the given fixtures will be assigned to all boards, this means that all boards set by current twister command can run those testcases which request the same fixtures.
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:

```yaml
- connected: false
  fixtures:
    - gpio_loopback
  id: 00063290670
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 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:

```yaml
- connected: false
  id: 0229000005d9ebc600000000000000000000000097969905
  platform: mimxrt1060_evk
  probe_id: 000609301751
  product: DAPLink CMSIS-DAP
  runner: jlink
  serial: null
```

Quarantine  Twister allows user to provide configuration files defining a list of tests or platforms 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. Multiple quarantine files can be used. 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.

When quarantining a class of tests or many scenarios in a single testsuite or when dealing with multiple issues within a subsystem, it is possible to use regular expressions, for example, kernel.* would quarantine all kernel tests.

An example of entries in a quarantine yaml:

```yaml
- scenarios:
   - sample.basic.helloworld
   comment: "Link to the issue: https://github.com/zephyrproject-rtos/zephyr/pull/33287"

- scenarios:
   - kernel.common
   - kernel.common.(misra|tls)
```

(continues on next page)
To exclude a platform, use the following syntax:

```yaml
- platforms:
  - qemu_x86
  comment: "broken qemu"
```

Additionally you can quarantine entire architectures or a specific simulator for executing tests.

**Test Configuration**

A test configuration can be used to customize various aspects of twister and the default enabled options and features. This allows tweaking the filtering capabilities depending on the environment and makes it possible to adapt and improve coverage when targeting different sets of platforms.

The test configuration also adds support for test levels and the ability to assign a specific test to one or more levels. Using command line options of twister it is then possible to select a level and just execute the tests included in this level.

Additionally, the test configuration allows defining level dependencies and additional inclusion of tests into a specific level if the test itself does not have this information already.

In the configuration file you can include complete components using regular expressions and you can specify which test level to import from the same file, making management of levels easier.

To help with testing outside of upstream CI infrastructure, additional options are available in the configuration file, which can be hosted locally. As of now, those options are available:

- Ability to ignore default platforms as defined in board definitions (Those are mostly emulation platforms used to run tests in upstream CI)
- Option to specify your own list of default platforms overriding what upstream defines.
- Ability to override `build_on_all` options used in some testcases. This will treat tests or sample as any other just build for default platforms you specify in the configuration file or on the command line.
- Ignore some logic in twister to expand platform coverage in cases where default platforms are not in scope.

**Platform Configuration**

The following options control platform filtering in twister:

- `override_default_platforms`: override default key a platform sets in board configuration and instead use the list of platforms provided in the configuration file as the list of default platforms. This option is set to False by default.
- `increased_platform_scope`: This option is set to True by default, when disabled, twister will not increase platform coverage automatically and will only build and run tests on the specified platforms.
- `default_platforms`: A list of additional default platforms to add. This list can either be used to replace the existing default platforms or can extend it depending on the value of `override_default_platforms`.

And example platforms configuration:
Test Level Configuration  The test configuration allows defining test levels, level dependencies and additional inclusion of tests into a specific test level if the test itself does not have this information already.

In the configuration file you can include complete components using regular expressions and you can specify which test level to import from the same file, making management of levels simple.

And example test level configuration:

```yaml
levels:
  - name: my-test-level
    description: >
      my custom test level
    adds:
      - kernel_threads.*
      - kernel_timer_behavior
      - arch_interrupt
      - boards.*
```

Combined configuration  To mix the Platform and level configuration, you can take an example as below:

An example platforms plus level configuration:

```yaml
platforms:
  override_defaultPlatforms: true
  default_platforms:
    - frdm_k64f
levels:
  - name: smoke
    description: >
      A plan to be used verifying basic zephyr features.
  - name: unit
    description: >
      A plan to be used verifying unit test.
  - name: integration
    description: >
      A plan to be used verifying integration.
  - name: acceptance
    description: >
      A plan to be used verifying acceptance.
  - name: system
    description: >
      A plan to be used verifying system.
  - name: regression
    description: >
      A plan to be used verifying regression.
```

To run with above test_config.yaml file, only default_platforms with given test level test cases will run.

Linux
Running in Tests in Random Order

Enable ZTEST framework's CONFIG_ZTEST_SHUFFLE config option to run your tests in random order. This can be beneficial for identifying dependencies between test cases. For native_sim platforms, you can provide the seed to the random number generator by providing -seed=value as an argument to twister. See *Shuffling Test Sequence* for more details.

Robot Framework Tests

Zephyr supports Robot Framework as one of solutions for automated testing.

Robot files allow you to express interactive test scenarios in human-readable text format and execute them in simulation or against hardware. At this moment Zephyr integration supports running Robot tests in the Renode simulation framework.

To execute a Robot test suite with twister, run the following command:

**Linux**

```
$ ./scripts/twister --platform hifive1 --test samples/subsys/shell/shell_module/sample.→shell.shell_module.robot
```

**Windows**

```
python ./scripts\twister --platform hifive1 --test samples/subsys/shell/shell_module/sample.→shell.shell_module.robot
```

It's also possible to run it by west directly, with:

```
$ ROBOT_FILES=shell_module.robot west build -p -b hifive1 -s samples/subsys/shell/shell_→module -t run_renode_test
```

Writing Robot tests For the list of keywords provided by the Robot Framework itself, refer to the official Robot documentation.

Information on writing and running Robot Framework tests in Renode can be found in the testing section of Renode documentation. It provides a list of the most commonly used keywords together with links to the source code where those are defined.

It's possible to extend the framework by adding new keywords expressed directly in Robot test suite files, as an external Python library or, like Renode does it, dynamically via XML-RPC. For details see the extending Robot Framework section in the official Robot documentation.

Running a single testsuite To run a single testsuite instead of a whole group of test you can run:

```
$ twister -p qemu_riscv32 -s tests/kernel/interrupt/arch.shared_interrupt
```

2.12.3 Twister blackbox tests

This guide aims to explain the structure of a test file so the reader will be able to understand existing files and create their own. All developers should fix any tests they break and create new ones when introducing new features, so this knowledge is important for any Twister developer.
Basics

Twister blackbox tests are written in python, using the pytest library. Read up on it [here](#). Auxiliary test data follows whichever format it was in originally. Tests and data are wholly contained in the `scripts/tests/twister_blackbox` directory and prepended with `test_`.

Blackbox tests should not be aware of the internal twister code. Instead, they should call twister as user would and check the results.

Sample test file

```python
#!/usr/bin/env python3
# Copyright (c) 2024 Intel Corporation
#
# SPDX-License-Identifier: Apache-2.0

import importlib
import mock
import os
import pytest
import sys
import json

from conftest import ZEPHYR_BASE, TEST_DATA, testsuite_filename_mock
from twisterlib.testplan import TestPlan

class TestDummy:
    TESTDATA_X = [
        ("smoke", 5),
        ("acceptance", 6),
    ]

    @classmethod
    def setup_class(cls):
        apath = os.path.join(ZEPHYR_BASE, "scripts", "twister")
        cls.loader = importlib.machinery.SourceFileLoader("__main__", apath)
        cls.spec = importlib.util.spec_from_loader(cls.loader.name, cls.loader)
        cls.twister_module = importlib.util.module_from_spec(cls.spec)

    @classmethod
    def teardown_class(cls):
        pass

    @pytest.mark.parametrize(
        "level, expected_tests",
        TESTDATA_X,
        ids=["smoke", "acceptance"]
    )
    @mock.patch.object(TestPlan, "TESTSUITE_FILENAME", testsuite_filename_mock)
    def test_level(self, capfd, out_path, level, expected_tests):
        # Select platforms used for the tests
        test_platforms = ["qemu_x86", "frdm_k64f"]
        # Select test root
        path = os.path.join(TEST_DATA, "tests")
        config_path = os.path.join(TEST_DATA, "test_config.yaml")

        # Set flags for our Twister command as a list of strs
        args = (  
            # Flags related to the generic test setup:
            # * Control the level of detail in stdout/err
            # * Establish the output directory
```

(continues on next page)
# * Select Zephyr tests to use
# * Control whether to only build or build and run aforementioned tests
["-i", "--outdir", out_path, "-T", path, "-y"]

# Flags under test
+ ["--level", level]
# Flags required for the test
+ ["--test-config", config_path]
# Flags related to platform selection
+ [

    for pair in zip("-p") * len(test_platforms), test_platforms)
    for val in pair
]

# First, provide the args variable as our Twister command line arguments.
# Then, catch the exit code in the sys_exit variable.
with mock.patch.object(sys, "argv", [sys.argv[0]] + args), pytest.raises(SystemExit) as sys_exit:
    # Execute the Twister call itself.
    self.loader.exec_module(self.twister_module)

# Check whether the Twister call succeeded
assert str(sys_exit.value) == "0"

# Access to the test file output
with open(os.path.join(out_path, "testplan.json")) as f:
    j = json.load(f)
filtered_j = [
    (ts["platform"], ts["name"], tc["identifier"])
    for ts in j["testsuites"]
    for tc in ts["testcases"]
    if "reason" not in tc
]

# Read stdout and stderr to out and err variables respectively
out, err = capfd.readouterr()
# Rewrite the captured buffers to stdout and stderr so the user can still read them
sys.stdout.write(out)
sys.stderr.write(err)

# Test-relevant checks
assert expected_tests == len(filtered_j)

---

Comparison with CLI

Test above runs the command

```
twister -i --outdir $OUTDIR -T $TEST_DATA/tests -y --level $LEVEL
--test-config $TEST_DATA/test_config.yaml -p qemu_x86 -p frdm_k64f
```

It presumes a CLI with the zephyr-env.sh or zephyr-env.cmd already run.

Such a test provides us with all the outputs we typically expect of a Twister run thanks to importlib's `exec_module()`¹. We can easily set up all flags that we expect from a Twister call via

¹ Take note of the `setup_class()` class function, which allows us to run `twister python file` as if it were called directly (bypassing the `__name__ == '__main__'` check).
We can check the standard output or stderr in \texttt{out} and \texttt{err} variables. Beside the standard outputs, we can also investigate the file outputs, normally placed in \texttt{twister-out} directories. Most of the time, we will use the \texttt{out_path} fixture in conjunction with \texttt{--outdir} flag (L52) to keep test-generated files in temporary directories. Typical files read in blackbox tests are \texttt{testplan.json}, \texttt{twister.xml} and \texttt{twister.log}.

### Other functionalities

#### Decorators

- @pytest.mark.usefixtures('clear_log')
  - allows us to use \texttt{clear_log} fixture from \texttt{conftest.py}. The fixture is to become autouse in the future. After that, this decorator can be removed.

- @pytest.mark.parametrize('level, expected_tests', TESTDATA_X, ids=['smoke', 'acceptance'])
  - this is an example of \texttt{pytest}'s test parametrization. Read up on it here. TESTDATAs are most often declared as class fields.

- @mock.patch.object(TestPlan, 'TESTSUITE_FILENAME', testsuite_filename_mock)
  - this decorator allows us to use only tests defined in the \texttt{test_data} and ignore the Zephyr testcases in the \texttt{tests} directory. Note that all “test data“ tests use \texttt{test_data.yaml} as a filename, not \texttt{testcase.yaml}! Read up on the mock library here.

#### Fixtures

Blackbox tests use \texttt{pytest}'s fixtures, further reading on which is available here. If you would like to add your own fixtures, consider whether they will be used in just one test file, or in many.

- If in many, create such a fixture in the \texttt{scripts/tests/twister_blackbox/conftest.py} file.
  - \texttt{scripts/tests/twister_blackbox/conftest.py} already contains some fixtures - take a look there for an example.

- If in just one, declare it in that file.
  - Consider using class fields instead - look at TESTDATAs for an example.

#### How do I...

**Call Twister multiple times in one test?** Sometimes we want to test something that requires prior Twister use. \texttt{--test-only} flag would be a typical example, as it is to be coupled with previous \texttt{--build-only} Twister call. How should we approach that?

If we just call the \texttt{importlib's exec_module} two times, we will experience log duplication. \texttt{twister.log} will duplicate every line (triplicate if we call it three times, etc.) instead of overwriting the log or appending to the end of it.

It is caused by the use of logger module variables in the Twister files. Thus us executing the module again causes the loggers to have multiple handles.

To overcome this, between the calls you ought to use

\footnote{\textsuperscript{2} We advise you to keep the first section of \texttt{args} definition intact in almost all of your tests, as it is used for the common test setup.}
2.12.4 Integration with pytest test framework

Please mind that integration of twister with pytest is still work in progress. Not every platform type is supported in pytest (yet). If you find any issue with the integration or have an idea for an improvement, please, let us know about it and open a GitHub issue/enhancement.

Introduction

Pytest is a python framework that “makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries” (https://docs.pytest.org/en/7.3.x/). Python is known for its free libraries and ease of using it for scripting. In addition, pytest utilizes the concept of plugins and fixtures, increasing its expendability and reusability. A pytest plugin pytest-twister-harness was introduced to provide an integration between pytest and twister, allowing Zephyr's community to utilize pytest functionality with keeping twister as the main framework.

Integration with twister

By default, there is nothing to be done to enable pytest support in twister. The plugin is developed as a part of Zephyr's tree. To enable install-less operation, twister first extends PYTHONPATH with path to this plugin, and then during pytest call, it appends the command with -p twister_harness.plugin argument. If one prefers to use the installed version of the plugin, they must add --allow-installed-plugin flag to twister's call.

Pytest-based test suites are discovered the same way as other twister tests, i.e., by a presence of testcase/sample.yaml. Inside, a keyword harness tells twister how to handle a given test. In the case of harness: pytest, most of twister workflow (test suites discovery, parallelization, building and reporting) remains the same as for other harnesses. The change happens during the execution step. The below picture presents a simplified overview of the integration.

If harness: pytest is used, twister delegates the test execution to pytest, by calling it as a subprocess. Required parameters (such as build directory, device to be used, etc.) are passed through a CLI command. When pytest is done, twister looks for a pytest report (results.xml) and sets the test result accordingly.

How to create a pytest test

An example folder containing a pytest test, application source code and Twister configuration .yaml file can look like the following:

test_foo/
    ├── pytest/
    │   └── test_foo.py
    ├── src/
    │   └── main.c
    │
    ├── CMakeList.txt
    │
    ├── prj.conf
    │
    └── testcase.yaml
2.12. Testing

Zephyr Project Documentation, Release 3.6.99

Test execution

Twister

Collecting tests (ba... Generation test conf... Applying filtration Spawn workers (paral... +

Building

Is pytest test?

yes

Run pytest with pytest-twister-harness plug...

no

Execute test directly in Twister with fi...

Twister

Collect test results

Generate reports

Text is not SVG - cannot display
An example of a pytest test is given at samples/subsys/testsuite/pytest/shell/pytest/test_shell.py. Using the configuration provided in the testcase.yaml file, Twister builds the application from src and then, if the .yaml file contains a harness: pytest entry, it calls pytest in a separate subprocess. A sample configuration file may look like this:

```
tests:
  some.foo.test:
    harness: pytest
    tags: foo
```

By default, pytest tries to look for tests in a pytest directory located next to a directory with binary sources. A keyword pytest_root placed under harness_config section in .yaml file can be used to point to other files, directories or subtests (more info here).

Pytest scans the given locations looking for tests, following its default discovery rules.

**Passing extra arguments**  There are two ways for passing extra arguments to the called pytest subprocess:

1. From .yaml file, using pytest_args placed under harness_config section - more info here.
2. Through Twister command line interface as --pytest-args argument. This can be particularly useful when one wants to select a specific testcase from a test suite. For instance, one can use a command:

```bash
$ ./scripts/twister --platform native_sim -T samples/subsys/testsuite/pytest/shell \
-s samples/subsys/testsuite/pytest/shell/sample.pytest.shell \n--pytest-args='--k test_shell_print_version'
```

**Fixtures**

dut  Give access to a DeviceAdapter type object, that represents Device Under Test. This fixture is the core of pytest harness plugin. It is required to launch DUT (initialize logging, flash device, connect serial etc). This fixture yields a device prepared according to the requested type (native, qemu, hardware, etc.). All types of devices share the same API. This allows for writing tests which are device-type-agnostic. Scope of this fixture is determined by the pytest_dut_scope keyword placed under harness_config section (more info here).

```python
from twister_harness import DeviceAdapter
def test_sample(dut: DeviceAdapter):
    dut.readlines_until('Hello world')
```

shell  Provide a Shell class object with methods used to interact with shell application. It calls wait_for_prompt method, to not start scenario until DUT is ready. The shell fixture calls dut fixture, hence has access to all its methods. The shell fixture adds methods optimized for interactions with a shell. It can be used instead of dut for tests. Scope of this fixture is determined by the pytest_dut_scope keyword placed under harness_config section (more info here).

```python
from twister_harness import Shell
def test_shell(shell: Shell):
    shell.exec_command('help')
```

mcumgr  Sample fixture to wrap mcumgr command-line tool used to manage remote devices. More information about MCUmgr can be found here MCUmgr.
**Note:** This fixture requires the `mcumgr` available in the system PATH

Only selected functionality of MCUmgr is wrapped by this fixture. For example, here is a test with a fixture `mcumgr`

```python
from twister_harness import DeviceAdapter, Shell, McuMgr

def test_upgrade(dut: DeviceAdapter, shell: Shell, mcumgr: McuMgr):
    # free the serial port for mcumgr
    dut.disconnect()
    # upload the signed image
    mcumgr.image_upload('path/to/zephyr.signed.bin')
    # obtain the hash of uploaded image from the device
    second_hash = mcumgr.get_hash_to_test()
    # test a new upgrade image
    mcumgr.image_test(second_hash)
    # reset the device remotely
    mcumgr.reset_device()
    # continue test scenario, check version etc.
```

### Classes

**DeviceAdapter**

A class defined in `twister_harness` used in tests to gather device output and send data to it.

```python
class twister_harness.DeviceAdapter(device_config: DeviceConfig)
    # This class defines a common interface for all device types (hardware, simulator, QEMU)
    # used in tests to gather device output and send data to it.

    launch() → None
        Start by closing previously running application (no effect if not needed). Then, flash
        and run test application. Finally, start an internal reader thread capturing an output
        from a device.

    connect() → None
        Connect to device - allow for output gathering.

    readline(timeout: float | None = None, print_output: bool = True) → str
        Read line from device output. If timeout is not provided, then use base_timeout.

    readlines(print_output: bool = True) → list[str]
        Read all available output lines produced by device from internal buffer.

    readlines_until(regex: str | None = None, num_of_lines: int | None = None, timeout: float
    | None = None, print_output: bool = True) → list[str]
        Read available output lines produced by device from internal buffer until following
        conditions:
        1. If regex is provided - read until regex regex is found in read line (or until timeout).
        2. If num_of_lines is provided - read until number of read lines is equal to
           num_of_lines (or until timeout).
        3. If none of above is provided - return immediately lines collected so far in internal
           buffer.

        If timeout is not provided, then use base_timeout.

    write(data: bytes) → None
        Write data bytes to device.
```
disconnect() → None
    Disconnect device - block output gathering.

close() → None
    Disconnect, close device and close reader thread.

Shell class `twister_harness.Shell`
    `device: DeviceAdapter, prompt: str = 'uart:~$', timeout: float | None = None`)

    Helper class that provides methods used to interact with shell application.

    `exec_command(command: str, timeout: float | None = None, print_output: bool = True) → list[str]`
        Send shell command to a device and return response. Passed command is extended by
double enter sings - first one to execute this command on a device, second one to re-
ceive next prompt what is a signal that execution was finished. Method returns print-
out of the executed command.

    `wait_for_prompt(timeout: float | None = None) → bool`
        Send every 0.5 second “enter” command to the device until shell prompt statement
will occur (return True) or timeout will be exceeded (return False).

Examples of pytest tests in the Zephyr project

- pytest_shell
- MCUmgr tests - tests/boot/with_mcumgr
- LwM2M tests - tests/net/lib/lwm2m/interop
- GDB stub tests - tests/subsys/debug/gdbstub

FAQ

How to flash/run application only once per pytest session?

dut is a fixture responsible for flashing/running application. By default, its scope is set
as function. This can be changed by adding to .yaml file pytest_dut_scope keyword
placed under harness_config section:

```
harness: pytest
harness_config:
    pytest_dut_scope: session
```

More info can be found here.

How to run only one particular test from a python file?

This can be achieved in several ways. In yaml file it can be added using a pytest_root
entry placed under harness_config with list of tests which should be run:

```
harness: pytest
harness_config:
    pytest_root:
    - "pytest/test_shell.py::test_shell_print_help"
```
Particular tests can be also chosen by `pytest -k` option (more info about `pytest` keyword filter can be found [here](#)). It can be applied by adding `-k` filter in `pytest_args` in `.yaml` file:

```
highlight
harness: pytest
harness_config:
  pytest_args:
    - "-k test_shell_print_help"
```

or by adding it to Twister command overriding parameters from the `.yaml` file:

```
$ ./scripts/twister ... --pytest-args='--k test_shell_print_help'
```

### How to get information about used device type in test?

This can be taken from `dut` fixture (which represents `DeviceAdapter` object):

```
highlight
device_type: str = dut.device_config.type
if device_type == 'hardware':
    ...
elif device_type == 'native':
    ...
```

### How to rerun locally pytest tests without rebuilding application by Twister?

This can be achieved by running Twister once again with `--test-only` argument added to Twister command. Another way is running Twister with highest verbosity level (`-vv`) and then copy-pasting from logs command dedicated for spawning `pytest` (log started by `Running pytest command: ...`).

### Is this possible to run pytest tests in parallel?

Basically `pytest-harness-plugin` wasn't written with intention of running `pytest` tests in parallel. Especially those one dedicated for hardware. There was assumption that parallelization of tests is made by Twister, and it is responsible for managing available sources (jobs and hardwares). If anyone is interested in doing this for some reasons (for example via `pytest-xdist` plugin) they do so at their own risk.

### Limitations

- Not every platform type is supported in the plugin (yet).

### 2.12.5 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`.
   ```bash
   west build -b mps2/an385 -- -DCONFIG_COVERAGE=y -DCONFIG_COVERAGE_DUMP=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:
   ```bash
   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:
   ```bash
   $ 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:
   ```bash
   $ find $ZEPHYR_SDK_INSTALL_DIR -iregex ".*gcov"
   ```

5. Create an output directory for the reports:
   ```bash
   $ mkdir -p gcov_report
   ```

6. Run gcovr to get the reports:
   ```bash
   $ gcovr -r $ZEPHYR_BASE -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:

```sh
west build -b native_sim 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:

```sh
$ twister --coverage -p qemu_x86 -T tests/kernel
```

or:

```sh
$ twister --coverage -p native_sim -T tests/bluetooth
```

which will produce `twister-out/coverage/index.html` report as well as the coverage data collected by `gcovr` tool in `twister-out/coverage.json`.

Other reports might be chosen with `--coverage-tool` and `--coverage-formats` command line options.

The process differs for unit tests, which are built with the host toolchain and require a different board:

```sh
$ twister --coverage -p unit_testing -T tests/unit
```

which produces a report in the same location as non-unit testing.

### Using different toolchains

Twister looks at the environment variable `ZEPHYR_TOOLCHAIN_VARIANT` to check which gcov tool to use by default. The following are used as the default for the Twister `--gcov-tool` argument default:

<table>
<thead>
<tr>
<th>Toolchain</th>
<th><code>-gcov-tool value</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>gcov</td>
</tr>
<tr>
<td>llvm</td>
<td>llvm-cov gcov</td>
</tr>
<tr>
<td>zephyr</td>
<td>gcov</td>
</tr>
</tbody>
</table>

### 2.12. Testing

255
2.12.6 BabbleSim

BabbleSim and Zephyr

In the Zephyr project we use the BabbleSim simulator to test some of the Zephyr radio protocols, including the BLE stack, 802.15.4, and some of the networking stack.

BabbleSim is a physical layer simulator, which in combination with the Zephyr bsim boards can be used to simulate a network of BLE and 15.4 devices. When we build Zephyr targeting a bsim board we produce a Linux executable, which includes the application, Zephyr OS, and models of the HW.

When there is radio activity, this Linux executable will connect to the BabbleSim Phy simulation to simulate the radio channel.

In the BabbleSim documentation you can find more information on how to get and build the simulator. In the nrf52_bsim and nrf5340bsim boards documentation you can find more information about how to build Zephyr targeting these particular boards, and a few examples.

Types of tests

Tests without radio activity: bsim tests with twister The bsim boards can be used without radio activity, and in that case, it is not necessary to connect them to a physical layer simulation. Thanks to this, these target boards can be used just like native_sim with twister, to run all standard Zephyr twister tests, but with models of a real SOC HW, and their drivers.

Tests with radio activity When there is radio activity, BabbleSim tests require at the very least a physical layer simulation running, and most, more than 1 simulated device. Due to this, these tests are not build and run with twister, but with a dedicated set of tests scripts.

These tests are kept in the tests/bsim/ folder. The compile.sh and run_parallel.sh scripts contained in that folder are used by the CI system to build the needed images and execute these tests in batch.

See sections below for more information about how to build and run them, as well as the conventions they follow.

There are two main sets of tests:

- Self checking embedded application/tests: In which some of the simulated devices applications are built with some checks which decide if the test is passing or failing. These embedded applications tests use the bs_tests system to report the pass or failure, and in many cases to build several tests into the same binary.

- Test using the EDTT tool, in which a EDTT (python) test controls the embedded applications over an RPC mechanism, and decides if the test passes or not. Today these tests include a very significant subset of the BT qualification test suite.

More information about how different tests types relate to BabbleSim and the bsim boards can be found in the bsim boards tests section.

Test coverage and BabbleSim

As the nrf52_bsim and nrf5340bsim boards are based on the POSIX architecture, you can easily collect test coverage information.

You can use the script tests/bsim/generate_coverage_report.sh to generate an html coverage report from tests.

Check the page on coverage generation for more info.
Building and running the tests

See the nrf52_bsim page for setting up the simulator.

The scripts also expect a few environment variables to be set. For example, from Zephyr’s root folder, you can run:

```bash
# Build all the tests
$ZEPHYR_BASE/tests/bsim/compile.sh

# Run them (in parallel)
RESULTS_FILE=$ZEPHYR_BASE/myresults.xml
SEARCH_PATH=$ZEPHYR_BASE/tests/bsim
$ZEPHYR_BASE/tests/bsim/run_parallel.sh
```

Or to build and run only a specific subset, e.g. host advertising tests:

```bash
# Build the Bluetooth host advertising tests
$ZEPHYR_BASE/tests/bsim/bluetooth/host/adv/compile.sh

# Run them (in parallel)
RESULTS_FILE=$ZEPHYR_BASE/myresults.xml
SEARCH_PATH=$ZEPHYR_BASE/tests/bsim/bluetooth/host/adv
$ZEPHYR_BASE/tests/bsim/run_parallel.sh
```

Check the run_parallel.sh help for more options and examples on how to use this script to run the tests in batch.

After building the tests’ required binaries you can run a test directly using its individual test script.

For example you can build the required binaries for the networking tests with

```bash
WORK_DIR=$ZEPHYR_BASE/bsim_out $ZEPHYR_BASE/tests/bsim/net/compile.sh
```

and then directly run one of the tests:

```bash
$ZEPHYR_BASE/tests/bsim/net/sockets/echo_test/tests_scripts/echo_test_802154.sh
```

Conventions

**Build scripts**  The build scripts compile.sh simply build all the required test and sample applications for the tests’ scripts placed in the subfolders below.

This build scripts use the common compile.source which provide a function (compile) which calls cmake and ninja with the provided application, configuration and overlay files.

To speed up compilation for users interested only in a subset of tests, several compile scripts exist in several subfolders, where the upper ones call into the lower ones.

Note that cmake and ninja are used directly instead of the west build wrapper as west is not required, and some Zephyr users do not use or have west, but still use the build and tests scripts.

**Test scripts**  Please follow the existing conventions and do not design one-off bespoke runners (e.g. a python script, or another shell abstraction).

The rationale is that it is easier and faster for the maintainers to perform tree-wide updates for build system or compatibility changes if the tests are run in the same manner, with the same variables, etc.
If you have a good idea for improving your test script, please make a PR changing all the test scripts in order to benefit everyone and conserve homogeneity. You can of course discuss it first in an RFC issue or on the babblesim discord channel.

Scripts starting with an underscore (_) are not automatically discovered and run. They can serve as either helper functions for the main script, or can be used for local development utilities, e.g. building and running tests locally, debugging, etc.

Here are the conventions:

- Each test is defined by a shell script with the extension .sh, in a subfolder called test_scripts/.
- It is recommended to run a single test per script file. It allows for better parallelization of the runs in CI.
- Scripts expect that the binaries they require are already built. They should not compile binaries.
- Scripts will spawn the processes for every simulated device and the physical layer simulation.
- Scripts must return 0 to the invoking shell if the test passes, and not 0 if the test fails.
- Each test must have a unique simulation id, to enable running different tests in parallel.
- Neither the scripts nor the images should modify the workstation filesystem content beyond the ${BSIM_OUT_PATH}/results/<simulation_id>/ or /tmp/ folders. That is, they should not leave stray files behind.
- Tests that require several consecutive simulations (e.g, if simulating a device pairing, powering off, and powering up after as a new simulation) should use separate simulation ids for each simulation segment, ensuring that the radio activity of each segment can be inspected a posteriori.
- Avoid overly long tests. If the test takes over 20 seconds of runtime, consider if it is possible to split it in several separate tests.
- If the test takes over 5 seconds, set EXECUTE_TIMEOUT to a value that is at least 5 times bigger than the measured run-time.
- Do not set EXECUTE_TIMEOUT to a value lower than the default.
- Tests should not be overly verbose: less than a hundred lines are expected on the outputs. Do make use of LOG_DBG() extensively, but don’t enable the DBG log level by default.

2.12.7 ZTest Deprecated APIs

Ztest is currently being migrated to a new API, this documentation provides information about the deprecated APIs which will eventually be removed. See Test Framework for the new API. Similarly, ZTest’s mocking framework is also deprecated (see Mocking via FFF).

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.

There exist two alternatives to writing tests. The first, and more verbose, approach is to directly declare and run the test suites. Here is a generic template for a test showing the expected use of ztest_test_suite():

```c
#include <zephyr/ztest.h>

extern void test_sometest1(void);
extern void test_sometest2(void);
ifndef CONFIG_WHATEVER
  /* Conditionally skip test_sometest3 */
extern void test_sometest3(void)
{
  ztest_test_skip();
}
#else
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);
}
```

Alternatively, it is possible to split tests across multiple files using ztest_register_test_suite() which bypasses the need for extern:

```c
#include <zephyr/ztest.h>

void test_sometest1(void) {
  zassert_true(1, "true");
}

ztest_register_test_suite(common, NULL,
                          ztest_unit_test(test_sometest1));
```
The above sample simple registers the test suite and uses a NULL pragma function (more on that later). It is important to note that the test suite isn’t directly run in this file. Instead two alternatives exist for running the suite. First, if to do nothing. A default test_main function is provided by ztest. This is the preferred approach if the test doesn’t involve a state and doesn’t require use of the pragma.

In cases of an integration test it is possible that some general state needs to be set between test suites. This can be thought of as a state diagram in which test_main simply goes through various actions that modify the board’s state and different test suites need to run. This is achieved in the following:

```c
#include <zephyr/ztest.h>

struct state {
    bool is_hibernating;
    bool is_usb_connected;
};

static bool pragma_always(const void *state) {
    return true;
}

static bool pragma_not_hibernating_not_connected(const void *s) {
    struct state *state = s;
    return !(state->is_hibernating && !state->is_usb_connected);
}

static bool pragma_usb_connected(const void *s) {
    return ((struct state *)s)->is_usb_connected;
}

ztest_register_test_suite(baseline, pragma_always, ztest_unit_test(test_case0));
ztest_register_test_suite(before_usb, pragma_not_hibernating_not_connected, ztest_unit_test(test_case1), ztest_unit_test(test_case2));
ztest_register_test_suite(with_usb, pragma_usb_connected, ztest_unit_test(test_case3), ztest_unit_test(test_case4));

void test_main(void) {
    struct state state;
    /* Should run `baseline` test suite only. */
    ztest_run_registered_test_suites(&state);
    /* Simulate power on and update state. */
    emulate_power_on();
    /* Should run `baseline` and `before_usb` test suites. */
    ztest_run_registered_test_suites(&state);
    /* Simulate plugging in a USB device. */
    emulate_plugging_in_usb();
    /* Should run `baseline` and `with_usb` test suites. */
    ztest_run_registered_test_suites(&state);
    /* Verify that all the registered test suites actually ran. */
    ztest_verify_all_registered_test_suites_ran();
}
```
For twister to parse source files and create a list of subcases, the declarations of ztest_test_suite() and ztest_register_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

```c
void test_main(void)
{
    #ifdef TEST_feature1
        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

```c
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():

```c
ztest_test_suite(common,
    ztest_unit_test(test_sometest1),
    ztest_unit_test(test_sometest2),
    /* 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

```c
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.

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:

```c
#include <zephyr/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)
{
    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 *

### 2.13 Static Code Analysis (SCA)

Support for static code analysis tools in Zephyr is possible through CMake.

The build setting `ZEPHYR_SCA_VARIANT` can be used to specify the SCA tool to use. `ZEPHYR_SCA_VARIANT` is also supported as an environment variable.

Use `-DZEPHYR_SCA_VARIANT=<tool>`, for example `-DZEPHYR_SCA_VARIANT=sparse` to enable the static analysis tool sparse.

#### 2.13.1 SCA Tool infrastructure

Support for an SCA tool is implemented in a file: `sca.cmake` file. The file: `sca.cmake` must be placed under file: `<SCA_ROOT>/cmake/sca/<tool>/sca.cmake`. Zephyr itself is always added as an SCA_ROOT but the build system offers the possibility to add additional folders to the SCA_ROOT setting.

You can provide support for out of tree SCA tools by creating the following structure:

```bash
<scaroot>/
  └── cmake/
  │   └── sca/<tool>/
```

# Custom SCA root

(continues on next page)
To add foo under /path/to/my_tools/cmake/sca create the following structure:

```
/path/to/my_tools
    └── cmake/
        └── sca/
            └── foo/
                └── sca.cmake
```

To use foo as SCA tool you must then specify `-DZEPHYR_SCA_VARIANT=foo`.
Remember to add `/path/to/my_tools` to `SCA_ROOT`.

`SCA_TOOL` can be set as a regular CMake setting using `-DSCA_ROOT=<sca_root>`, or added by a Zephyr module in its `module.yml` file, see [Zephyr Modules - Build settings](#).

## 2.13.2 Native SCA Tool support

The following is a list of SCA tools natively supported by Zephyr build system.

**CodeChecker support**

CodeChecker is a static analysis infrastructure. It executes analysis tools available on the build system, such as Clang-Tidy, Clang Static Analyzer and Cppcheck. Refer to the analyzer's websites for installation instructions.

### Installing CodeChecker

CodeChecker itself is a python package available on [pypi](https://pypi.org).

```
pip install codechecker
```

### Running with CodeChecker

To run CodeChecker, `west build` should be called with a `-DZEPHYR_SCA_VARIANT=codechecker` parameter, e.g.

```
w west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker
```

### Configuring CodeChecker

To configure CodeChecker or analyzers used, arguments can be passed using the `CODECHECKER_ANALYZE_OPTS` parameter, e.g.

```
w west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker \ -DCODECHECKER_ANALYZE_OPTS="--config;$CODECHECKER_CONFIG_FILE;--timeout;60"
```

### Storing CodeChecker results

If a CodeChecker server is active the results can be uploaded and stored for tracking purposes. Storing is done using the optional `CODECHECKER_STORE=y` or `CODECHECKER_STORE_OPTS="arg;list"` parameters, e.g.

```
w west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker \ -DCODECHECKER_STORE_OPTS="--name;build;--url=localhost:8001/Default"
```
Note: If --name isn't passed to either CODECHECKER_ANALYZE_OPTS or CODECHECKER_STORE_OPTS, the default zephyr is used.

Exporting CodeChecker reports  Optional reports can be generated using the CodeChecker results, when passing a -DCODECHECKER_EXPORT=<type> parameter. Allowed types are: html, json, codeclimate, gerrit, baseline. Multiple types can be passed as comma-separated arguments.

Optional parser configuration arguments can be passed using the CODECHECKER_PARSE_OPTS parameter, e.g.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=codechecker 
-DCODECHECKER_EXPORT=html,json -DCODECHECKERPARSE_OPTS="--trim-path-prefix;$PWD"
```

Sparse support

Sparse is a static code analysis tool. Apart from performing common code analysis tasks it also supports an address_space attribute, which allows introduction of distinct address spaces in C code with subsequent verification that pointers to different address spaces do not get confused. Additionally it supports a force attribute which should be used to cast pointers between different address spaces. At the moment Zephyr introduces a single custom address space __cache used to identify pointers from the cached address range on the Xtensa architecture. This helps identify cases where cached and uncached addresses are confused.

Running with sparse  To run a sparse verification build west build should be called with a -DZEPHYR_SCA_VARIANT=sparse parameter, e.g.

```
west build -d hello -b intel_adsp/cavs25 zephyr/samples/hello_world -- -DZEPHYR_SCA-_VARIA
T=sparse
```

GCC static analysis support

Static analysis was introduced in GCC 10 and it is enabled with the option -fanalyzer. This option performs a much more expensive and thorough analysis of the code than traditional warnings.

Run GCC static analysis  To run GCC static analysis, west build should be called with a -DZEPHYR_SCA_VARIANT=gcc parameter, e.g.

```
wes t build -b qemu_x86 samples/userspace/hello_world_user -- -DZEPHYR_SCA_VARIAN T=gcc
```

Parasoft C/C++test support

Parasoft C/C++test is a software testing and static analysis tool for C and C++. It is a commercial software and you must acquire a commercial license to use it.

Documentation of C/C++test can be found at https://docs.parasoft.com/. Please refer to the documentation for how to use it.
Generating Build Data Files  To use C/C++test, `cpptestscan` must be found in your `PATH` environment variable. And `west build` should be called with a `-DZEPHYR_SCA_VARIANT=cpptest` parameter, e.g.

```
west build -b qemu_cortex_m3 zephyr/samples/hello_world -- -DZEPHYR_SCA_VARIANT=cpptest
```

A `.bdf` file will be generated as `build/sca/cpptest/cpptestscan.bdf`.

Generating a report file  Please refer to Parasoft C/C++test documentation for more details. To import and generate a report file, something like the following should work.

```
cpptestcli -data out -localsettings local.conf -bdf build/sca/cpptest/cpptestscan.bdf -
-config "builtin://Recommended Rules" -report out/report
```

You might need to set `bdf.import.c.compiler.exec`, `bdf.import.cpp.compiler.exec`, and `bdf.import.linker.exec` to the toolchain `west build` used.

## 2.14 Toolchains

Guides on how to set up toolchains for Zephyr development.

### 2.14.1 Zephyr SDK

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 and OpenOCD.

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

**Supported architectures**

The Zephyr SDK supports the following target architectures:

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

**Installation bundle and variables**

The Zephyr SDK bundle supports all major operating systems (Linux, macOS and Windows) and is delivered as a compressed file. The installation consists of extracting the file and running the included setup script. Additional OS-specific instructions are described in the sections below.

If no toolchain is selected, the build system looks for Zephyr SDK and uses the toolchain from there. You can enforce this by setting the environment variable `ZEPHYR_TOOLCHAIN_VARIANT` to `zephyr`.

### 2.14. Toolchains
If you install the Zephyr SDK outside any of the default locations (listed in the operating system specific instructions below) and you want automatic discovery of the Zephyr SDK, then you must register the Zephyr SDK in the CMake package registry by running the setup script. If you decide not to register the Zephyr SDK in the CMake registry, then the ZEPHYR_SDK_INSTALL_DIR can be used to point to the Zephyr SDK installation directory.

You can also set ZEPHYR_SDK_INSTALL_DIR to point to a directory containing multiple Zephyr SDKs, allowing for automatic toolchain selection. For example, you can set ZEPHYR_SDK_INSTALL_DIR to /company/tools, where the company/tools folder contains the following subfolders:

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

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

Zephyr SDK version compatibility

In general, the Zephyr SDK version referenced in this page should be considered the recommended version for the corresponding Zephyr version.

For the full list of compatible Zephyr and Zephyr SDK versions, refer to the Zephyr SDK Version Compatibility Matrix.

Zephyr SDK installation

Note: You can change 0.16.5-1 to another version in the instructions below if needed; the Zephyr SDK Releases page contains all available SDK releases.

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

Ubuntu

1. Download and verify the Zephyr SDK bundle:

   cd ~
   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
   wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/sha256.sum | shasum --check --ignore-missing

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

2. Extract the Zephyr SDK bundle archive:

   tar xvzf zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz

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

   • $HOME
   • $HOME/.local
3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.5-1
./setup.sh
```

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

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

```
sudo cp ~/zephyr-sdk-0.16.5-1/sysroots/x86_64-pokysdk-linux/usr/share/openocd/
contrib/60-openocd.rules /etc/udev/rules.d
```

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

**macOS**

1. Download and verify the Zephyr SDK bundle:

```
cd ~
curl -L -O https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/zephyr-sdk-0.16.5-1_macos-x86_64.tar.xz
curl -L https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/sha256.sum | sha256sum --check --ignore-missing
```

If your host architecture is 64-bit ARM (Apple Silicon, also known as M1), replace x86_64 with aarch64 in order to download the 64-bit ARM macOS SDK.

2. Extract the Zephyr SDK bundle archive:

```
tar xvf zephyr-sdk-0.16.5-1_macos-x86_64.tar.xz
```

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

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

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

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.5-1
./setup.sh
```
Zephyr Project Documentation, Release 3.6.99

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

**Windows**

1. Open a cmd.exe terminal window as a regular user
2. Download the Zephyr SDK bundle:
   ```
cd %HOMEPATH%
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.16.5-1/zephyr-sdk-0.16.5-1_windows-x86_64.7z
   ``
3. Extract the Zephyr SDK bundle archive:
   ```
7z x zephyr-sdk-0.16.5-1_windows-x86_64.7z
   ```

   **Note:** It is recommended to extract the Zephyr SDK bundle at one of the following locations:
   - %HOMEPATH%
   - %PROGRAMFILES%

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

4. Run the Zephyr SDK bundle setup script:
   ```
cd zephyr-sdk-0.16.5-1
setup.cmd
   ```

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

**2.14.2 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
   ```
# Windows, license file:

```bash
set ARM_LICENSE_FILE=c:\<path>\license_armds.dat
```

# Windows, license server:

```bash
set ARM_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.

## 2.14.3 Cadence Tensilica Xtensa C/C++ Compiler (XCC)

1. Obtain Tensilica Software Development Toolkit targeting the specific SoC on hand. This usually contains two parts:
   - The Xtensa Xplorer which contains the necessary executables and libraries.
   - A SoC-specific add-on to be installed on top of Xtensa Xplorer.
     - This add-on allows the compiler to generate code for the SoC on hand.

2. Install Xtensa Xplorer and then the SoC add-on.
   - Follow the instruction from Cadence on how to install the SDK.
   - Depending on the SDK, there are two set of compilers:
     - GCC-based compiler: xt-xcc and its friends.
     - Clang-based compiler: xt-clang and its friends.

3. Make sure you have obtained a license to use the SDK, or has access to a remote licensing server.

4. Set these environment variables:
   - Set `ZEPHYR_TOOLCHAIN_VARIANT` to xcc or xt-clang.
   - Set `XTENSA_TOOLCHAIN_PATH` to the toolchain installation directory.
   - Set `XTENSA_CORE` to the SoC ID where application is being targeting.
   - Set `TOOLCHAIN_VER` to the Xtensa SDK version.

5. For example, assuming the SDK is installed in `/opt/xtensa`, and using the SDK for application development on `intel_adsp_cavs15`, setup the environment using:

```bash
# Linux
export ZEPHYR_TOOLCHAIN_VARIANT=xcc
export XTENSA_TOOLCHAIN_PATH=/opt/xtensa/XtDevTools/install/tools/
export XTENSA_CORE=X6H3SUE_RI_2018_0
export TOOLCHAIN_VER=RI-2018.0-linux
```

6. To use Clang-based compiler:
• Set `ZEPHYR_TOOLCHAIN_VARIANT` to `xt-clang`.

• Note that the Clang-based compiler may contain an old LLVM bug which results in the following error:

```
/tmp/file.s: Assembler messages:
/tmp/file.s:20: Error: file number 1 already allocated
clang-3.9: error: Xtensa-as command failed with exit code 1
```

If this happens, set `XCC_NO_G_FLAG` to 1.

  – For example:

```
# Linux
export XCC_NO_G_FLAG=1
```

### 2.14.4 DesignWare ARC MetaWare Development Toolkit (MWDT)

1. You need to have ARC MWDT installed on your host.

2. You need to have Zephyr SDK installed on your host.

**Note:** A Zephyr SDK is used as a source of tools like device tree compiler (DTC), QEMU, etc... Even though ARC MWDT toolchain is used for Zephyr RTOS build, still the GNU preprocessor & GNU objcopy might be used for some steps like device tree preprocessing and `.bin` file generation. We used Zephyr SDK as a source of these ARC GNU tools as well.

3. **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).

**Tip:** If you have only one ARC MWDT toolchain version installed on your machine you may skip setting `ARCMWDT_TOOLCHAIN_PATH` - it would be detected automatically.

4. 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_2023.03/

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
arcmwdt
> echo %ARCMWDT_TOOLCHAIN_PATH%
C:\ARC\MWDT_2023.03\```
2.14.5 GNU Arm Embedded

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 for this guide that you install into the directory `C:\gnu_arm_embedded`. You can also choose the default installation path used by the ARM GCC installer, in which case you will need to adjust the path accordingly in the guide below.

**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`)

2.14.6 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`.

### 2.14.7 Crosstool-NG (Deprecated)

**Warning:** xtools toolchain variant is deprecated. The cross-compile toolchain variant should be used when using a custom toolchain built with 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:
   ```bash
   ./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):
   ```bash
   # Linux, macOS:
   $ echo $ZEPHYR_TOOLCHAIN_VARIANT
   xtools
   $ echo $XTOOLS_TOOLCHAIN_PATH
   /Volumes/CrossToolNGNew/build/output/
   ```

### 2.14.8 Host Toolchains

In some specific configurations, like when building for non-MCU x86 targets on a Linux host, you may be able to reuse 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`.

### 2.14.9 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 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-.

### 2.14.10 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 Device-tree 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/modules/FindHostTools.cmake and cmake/modules/FindTargetTools.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=...
```

### 2.15 Tools and IDEs

#### 2.15.1 CLion

**CLion** is a cross-platform C/C++ IDE that supports multi-threaded RTOS debugging.

This guide describes the process of setting up, building, and debugging Zephyr’s multi-thread-blinky sample in CLion.

The instructions have been tested on Windows. In terms of the CLion workflow, the steps would be the same for macOS and Linux, but make sure to select the correct environment file and to adjust the paths.

**Get CLion**

Download CLion and install it.

**Initialize a new workspace**

This guide gives details on how to build and debug the multi-thread-blinky sample application, but the instructions would be similar for any Zephyr project and `workspace layout`.

Before you start, make sure you have a working Zephyr development environment, as per the instructions in the **Getting Started Guide**.