User's Guide

Getting Started with Zephyr on MSPM0 Microcontrollers

ABSTRACT

This guide provides a high-level entry point for users who want to use the open-source Zephyr Real Time Operating System (RTOS) using MSPM0 microcontrollers (MCUs). It highlights the software resources available for Texas Instruments MSPM0, explains the features of MSPM0 devices, and outlines the following tools: OpenOCD, VSCode debugging, and GNU Debugging.

This guide covers:

- What is Zephyr
- Why use Zephyr with Texas Instruments MSPM0 Microcontrollers
- How to set up the Zephyr environment
- Differences between the Zephyr upstream repository and the Texas Instruments maintained downstream repository
- Overview of debugging Zephyr projects using the command line and Visual Studio Code.
- How to run examples on a MSPM0 LaunchPad™

Table of Contents

Trademarks

1 What is Zephyr?

1.1 Real Time Operating System (RTOS)

A real-time operating system is a type of computer operating system that is well-suited to embedded applications. RTOSes are designed to be small and deterministic, with low overhead to allow for time-sensitivity when executing commands.

RTOSes, while being significantly lighter weight than an operating system like Linux, still have many of the benefits of multitasking, scheduling, and memory handling. With these features, RTOSes are designed to be great for complex applications, and still be memory, power, and compute sensitive.

There are many great resources available for understanding the complexities of multitasking, scheduling, and memory architecture, which will not be covered here. However, for the ability to get up and running with Zephyr, it is important to have a basic understanding of the following RTOS concepts:

- Multitasking/threading: Operating Systems employ a kernel, which is the core process that allows multiple “users” or programs to access a processor’s computing and memory resources. This simultaneous operation is accomplished through threading, where each program running is assigned a thread (or a task in Zephyr). These tasks can run “concurrently” when combined with scheduling.
- Scheduling: The core component of the kernel, which allows the illusion of simultaneous execution, is scheduling. The scheduler is a fundamental block of code within the operating system that has the ability to pause, resume, and switch tasks multiple times during the execution of the task. This means that when one task has downtime, or is not actively executing code, then a different task can take control of the compute and memory until it is done, or a task with a higher priority begins.
- Real-Time Operation: In many applications, real-time response to external stimuli is required. This is the difference between RTOSes like Zephyr as compared to typical, non-real-time operating systems like Linux. When tasks are created, they are assigned a priority, similar to an interrupt. Tasks that require a time-sensitive response are assigned a higher priority than non-time-sensitive tasks, which allows them to use the compute as soon as it is created.

1.2 Zephyr as an Open-Source RTOS Option

Zephyr is an open-source RTOS that has been growing in popularity among the embedded development sphere over the last decade. Zephyr is a community-maintained RTOS that does not require royalties for a license.

2 Benefits of Zephyr on MSPM0

2.1 Advantages over Bare Metal

Typically, running multiple processes or tasks on a single application can be quite challenging, requiring complex memory management from the software engineer. However, RTOSes simplify this process via the kernel. Despite having higher overhead than a typical bare-metal code, RTOS kernels like Zephyr are still great for memory-sensitive applications due to the relatively low overhead and compile-time memory optimizations. Another important consideration with Zephyr is the ease of code verification and security. As all application code in Zephyr is operating above the hardware abstraction layer, verification becomes much simpler. The software developer knows the foundation for the code is sound and only must truly worry themselves with the application code, which may be easier to debug due to the additional debugging and kernel features provided by Zephyr's kernel.

2.2 MSPM0 Considerations

The MSPM0 family of Arm Cortex-M0+ microcontrollers excel when it comes to power, size, and cost. Due to this, MSPM0 is applicable to most every application, with the ability to integrate analog, communication, and housekeeping features into one package. Alongside Zephyr's lightweight and time-sensitive properties, the MSPM0 becomes the perfect operating system for a wide variety of embedded projects.

Zephyr contains the following properties, which make it beneficial for power, memory, and getting the maximum performance out of the MSPM0.

- Power:
  - Idle operations are tickless, saving energy when in low-power mode
  - MCU can enter deep-sleep state when no tasks are running
- Memory:
  - Zephyr's kernel fits within a few KB of ROM and RAM, working well with the MSPM0's smaller memory resources
- Performance:
  - Pre-emptive scheduler provides predictable task latencies
  - Ideal for applications like sensor fusion or motor control due to the tight real-time performance
  - Threading/Tasking allows for making full usage of the CPU's capabilities, as tasks are only running when necessary

Overall, Zephyr complements the MSPM0 family of devices' low-power and cost-effective advantages by enabling a scalable, standards-based RTOS environment that adds multitasking while preserving the device's efficiency and deterministic behavior.

2.3 Common Applications

Oftentimes, engineers will use an RTOS in time-sensitive applications such as medical, industrial, automotive, and wearable applications. Reviewing the advantages of Zephyr makes it clear how it may be useful in these situations. The time-sensitive properties work well in a medical, industrial, or automotive application where immediate response is critical to user safety and application consistency.

The Zephyr stack has numerous TI and third-party sensors integrated, making it simple for the designer to quickly bring up a full system or prototype without needing to code these by hand. Additionally, Zephyr is open-source, so updated sensors are added frequently. Battery chargers, connectivity solutions, environmental sensors, and more are available to be interfaced with immediately.

2.4 Security Overview

Across all of these applications, security has become an important consideration regardless of the application. While operating systems provide a natural layer of application security through the increased layers of abstraction, Zephyr takes additional steps to improve full-system security. A critical component of security in embedded applications is memory protection, and in bare metal applications, this is largely left to the software developer. Zephyr on the other hand offers isolation, meaning a small coding mistake is less likely to have system-wide consequences. This, however doesn't mean that Zephyr provides full protection from attacks, and it is important to follow secure coding guidelines during development to minimize the attack vectors within Zephyr.

While this is a brief overview of Zephyr's security features, a full overview of Zephyr's security features can be found in their documentation.

3 How to set up a Zephyr Development Environment

3.1 General Setup

Zephyr is installed differently based on the user's operating system. For this document, the installation will be shown in Ubuntu 22.04 LTS. For Windows and MacOS, please consult the guide. However, much of the installation is performed using Python and west, meaning there are significantly fewer differences in the installation process between operating systems.

3.1.1 Installing Dependencies

At the time of writing, there are four requirements for installing Zephyr:

- CMake Version 3.20.5 or above
- Python Version 3.10 or above
- Devicetree compiler Version 1.4.6 or above
- Git

3.1.2 Setting up Python and Zephyr

Zephyr has quite a few Python dependencies, and it is important to keep these updated consistently. For first time setup, follow these steps:

1. Install dependencies in Ubuntu with the following command:

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

2. Create a new virtual environment in the Home Path. This should be done for every new Zephyr project. "zephyrproject" is what will be used for this example, but this name can be freely changed as long as any future command is updated as well.

```
python -m venv ~/zephyrproject/.venv #create script for virtual environment
source ~/zephyrproject/.venv/bin/activate #activate virtual environment
```

3. Initialize west inside of a zephyrproject and cd into it. Run west update, this may take around 15 minutes, as many packages need to be installed.

```
pip install wheel
pip install west
west init ~/zephyrproject
cd ~/zephyrproject
west update
west zephyr-export
west packages pip --install
```

4. Finally, install the Zephyr SDK.
   deactivate

   ```
   cd ~

   git clone https://github.com/openocd-org/openocd.git

   sudo apt install libusb-1.0-0-dev libhidapi-dev
   ```

3.1.3 OpenOCD

Zephyr includes a version of OpenOCD as of Zephyr 0.17.4 that **does not** support TI LaunchPads. As such, it is critical to install a newer version that supports TI MSP MCUs.

If the guide has been followed up to this point, it is important to first deactivate the virtual environment that was used up to this point before creating a new folder for OpenOCD in the home directory.

```
deactivate

cd ~

git clone https://github.com/openocd-org/openocd.git

sudo apt install libusb-1.0-0-dev libhidapi-dev
```

Once this is completed, OpenOCD can be built within the OpenOCD folder. This build must be referenced during flashing for all builds on TI hardware.

```
cd <cloned_OPENOCD_dir>

git submodule update --init --recursive

cd jimtcl

./configure

make

sudo make install

cd .. #back in the cloned directory

sudo apt-get install libusb-1.0-0-dev

./bootstrap #when building from the git repository

./configure --enable-xds110 #optionally add any other debuggers as needed

make

sudo make install
```

OpenOCD is now ready to go for the following steps.

3.1.4 Differentiating the TI Downstream

The final step is to set up the TI downstream Zephyr repository. TI maintains a separate fork of the community-maintained Zephyr upstream. This is referred to as the TI downstream, the TI-maintained fork, which includes features that have not yet been included in the upstream. This includes additional boards, user-specific applications, new drivers, etc. Additionally, as this fork is actively maintained by TI, users can use the E2E forms for support. For items that are present in both the upstream and downstream, it is recommended to use the upstream fork.

The first step is to check the current status of the branch to see what the remote pointers are currently named.

```
cd ~/zephyrproject/zephyr

git status

git remote -v
```

If the previous steps were followed, then the upstream is installed and tracked. The output of the above commands should look something like this:

```
origin  https://github.com/zephyrproject-rtos/zephyr (fetch)

origin  https://github.com/zephyrproject-rtos/zephyr (push)
```

Origin is a nonspecific name, and as such, for clarity, it is recommended to refer to upstream and downstream explicitly. The following commands both rename the zephyr origin to upstream, while creating the source for TI's downstream.

```
git remote rename origin upstream

git remote add downstream https://github.com/TexasInstruments/msp-zephyr.git
```

The expected output after running git remote -v should be as follows:

```
#expected output after git remote -v

downstream      https://github.com/TexasInstruments/msp-zephyr.git (fetch)

downstream      https://github.com/TexasInstruments/msp-zephyr.git (push)

upstream        https://github.com/zephyrproject-rtos/zephyr (fetch)

upstream        https://github.com/zephyrproject-rtos/zephyr (push)
```

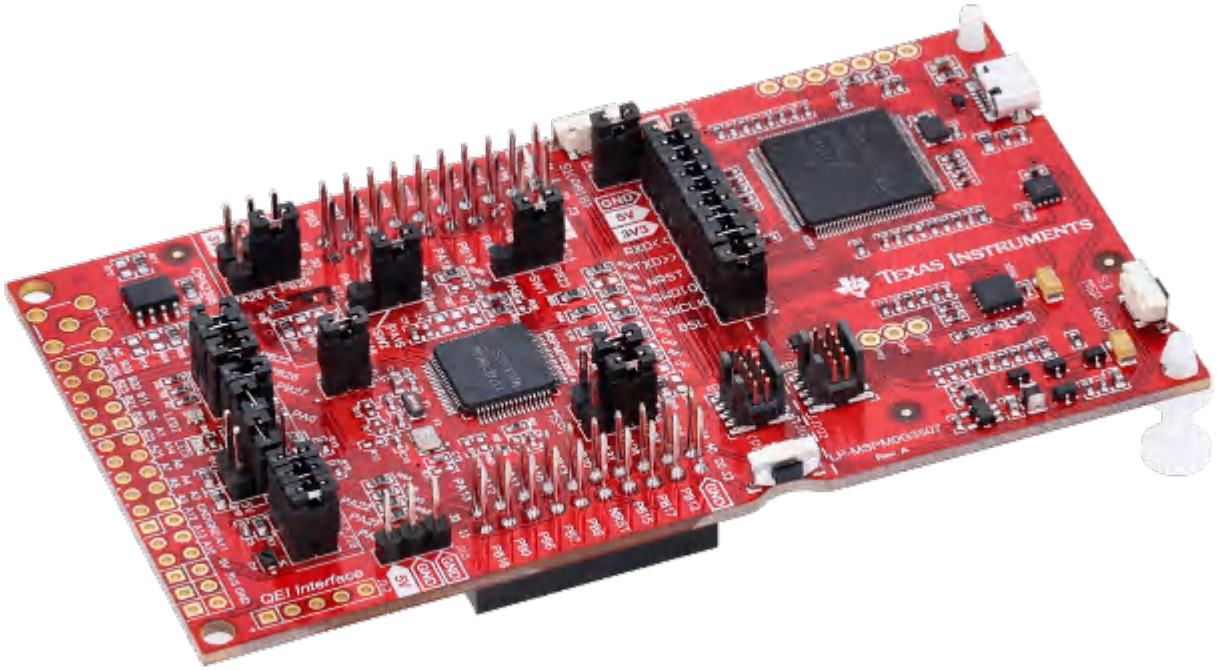
From there, any updates to the downstream can be accessed with the pointer downstream. Below is an example of checking out an existing branch and creating a new local branch for version management as new projects are developed.

```
git fetch downstream # receives information about the downstream

git checkout downstream/stable # checks out remote downstream stable branch without
creating local branch

git checkout -b stable downstream/stable # creates new local branch called stable that
tracks the downstream stable branch
```

4 How to Run Examples on an MSPM0 Launchpad

4.1 MSPM0 Launchpads

The MSPM0 family of devices has evaluation modules, named Launchpads, which can be used to get up and running with Zephyr testing and prototyping. Below is the LP-MSPM0G3507, which will be used for the following tutorial.



4.2 Running Projects on MSPM0 Launchpads

The following sections are based on the getting started guide from the Zephyr documentation. However, when there are important changes, they will be mentioned here.

4.2.1 Running Blinky

Once the file system is properly set up, connect a LaunchPad via USB. In the virtual environment, re-enter the virtual environment, cd into the /zephyr folder, and run the following:

```
cd ~/zephyrproject/zephyr
```

```
west build -p always -b lp_mspm0g3507 -d out samples/basic/blinky
```

```
west flash -d out --openocd ~/openocd/src/openocd --openocd-search ~/openocd/tcl
```

This builds and flashes the board using the OpenOCD version, which was installed earlier.

To see which boards are currently supported, run west build, and use the board name that corresponds to the LaunchPad. For the LP-MSPM0G3507 LaunchPad™, the board name is lp_mspm0g3507.

When the flashing completes, the LED will blink periodically upon a hardware restart. When flashing, the board enters debug mode and allows for debugging through west using the west debug command, or other debug tools as described in Section 5.2.3 Debugging Projects.

4.2.2 Running More Complex Examples

There are more complex examples covering the other MSPM0 peripherals, allowing the user to rapidly prototype with these prebuilt examples or modify them to use as a starting point for larger, more complex projects.

Similarly to blinky, to immediately run these examples, west build will build the project and west flash can be used to flash it to the board. Zephyr includes a wide variety of samples built for different boards, and due to Zephyr's portable design, there are many that can be easily ported to the MSPM0. These examples can be found in the samples/ folder, with a varying range of complexity.

4.3 Debugging Projects

4.3.1 GNU Debugger (GDB) with Command Line

GDB is a command-line interface that can be used to interface with the XDS-110 emulator to debug Zephyr projects. Similarly to CCS, the zephyr.elf file created when building a Zephyr project will allow for debugging using GDB.

There are multiple options for west debug; however, it is suggested that the reader reads through the Zephyr Project's west debug guide for any specific runners or debug tools that they may use.

4.3.2 Setting up Visual Studio Code (VSCode) Environment

Visual Studio Code is a common way to debug embedded projects, with the right extensions.

The extension Cortex-Debug allows visual debugging within Visual Studio Code. After installing the extension via the "extensions menu" within VSCode, the launch.json file must be configured to support MSPM0 Zephyr Projects. After this, the environment is ready to build, flash, and debug a Zephyr project.

```
{
    "version": "2.0.0",
    "configurations": [
        {
            "name": "Zephyr Debug",
            "executable":"<absolute_path_to>/zephyrproject/zephyr/out/zephyr/zephyr.elf,
            "request": "launch",
            "type": "cortex-debug",
            "runToEntryPoint": "main",
            "servertype": "external",
            "gdbPath": "<absolute_path_to>/<path_to_zephyr_sdk>/arm-zephyr-eabi/bin/arm-
zephyr-eabi-gdb",
            "gdbTarget": "localhost:3333",
            "device": "MSPM0G3507"
        },
    ]
}
```

4.3.3 Debugging using Cortex-Debug in VSCode

In VSCode, connect to the board using the board's .ccxml (can be found in most examples under the "target configuration" section), and begin project-less debugging using it. From there, zephyr.elf, which is generated by west when using the west build command, can be used to load the symbols.

Assuming the VSCode environment has been set up with debug-cortex as shown in Section 4.3.2, the following command can be used to start a debug server and run Start Debugging within VSCode.

```
west debugserver -d out --openocd ~/openocd/src/openocd \
--openocd-search ~/openocd/tcl
```

4.4 Creating your own project

New projects can either be built off of existing examples or created by creating a new folder in either the zephyrproject/zephyrfolder (repository app), the zephyrproject/ folder itself (workspace app), or outside of the previously created folders entirely (freestanding app). More information can be found on these different application types within the Zephyr docs.

For creating unique projects, the application development section of the Zephyrproject covers all that is needed to know. In general, creating a project based on one of the TI or Zephyr provided examples is an easier path to creating a project, as these have the necessary configuration files included by default, which means less finicking with overlays, prj.conf files, and so forth.

5 References

- RTOS Fundamentals Guide
- LP-MSPM0G3507 LaunchPad
- Zephyr Documents Getting Started Guide
- Cortex-Debug VSCode Extension

6 E2E

For support on TI products, use the E2E support forms to see frequently asked questions, and post new ones.

7 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

DATE	REVISION	NOTES
February 2026	*	Initial Release

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025