Zephyr Development Environment for Students

Johannes Witzig[1], Flavio Felder[1], Vixay Phimmasane[1], Simon Künzli[1*] and Andreas Rüst[1]

[1]*Institute of Embedded Systems, Zurich University of Applied Sciences, 8401 Winterthur, Switzerland*

Abstract

The Institute of Embedded Systems at the Zurich University of Applied Sciences uses Zephyr both in research projects and for several lab exercises for BSc, MSc, and students in continuing education. These labs include topics like embedded security, bootloaders, and general operating system concepts. In this paper, we discuss the challenge of providing a uniform Zephyr development environment that can be set up quickly and reliably.

Keywords

container, development environment, education, WSL, Zephyr, ZHAW

1. Introduction

At the Institute of Embedded Systems (InES), as an institute at a university of applied sciences, we have two main areas of activity. On the one hand, we teach in programs for undergraduates (bachelor's) and graduates (master's) and in continuing education. On the other hand, we work in research projects that are either publicly funded or funded by industry partners. Teaching about Real-time operating systems and their use is an essential topic in our electrical engineering, computer science, and systems engineering curricula. Zephyr is ideally suited for this purpose as it is open source, has momentum, and includes various communication stacks that enable the creation of small student projects with reasonable effort. In research projects, we need our firmware to support many different microcontrollers, transceivers, and sensors.

At our institute, we have been working with Zephyr since version 1.14 (2019). After a first successful evaluation of the RTOS in a Master's thesis, we started using it in research projects of various kinds [1], [2]. At the same time, we have started introducing Zephyr into classes through lecture content, laboratories, and student projects like Bachelor's theses.

As the students only work with Zephyr for a few hours, the effort of setting up a native Zephyr development environment on all the students' laptops seemed too high. In order to lower this effort for students to get a working Zephyr development environment, we discussed options. Initially, we used a standard Ubuntu desktop virtual machine (VM) to provide a development environment to the students. After testing this VM for one semester, we identified the following problems:

- Not reproducible
- Time-consuming to set up and maintain
- Large ($\approx$ 10 GiB)
- Uses a lot of resources (storage, RAM, CPU)
- Forces students to work inside the VM
- File sharing is unreliable
- Prompts the user to install updates

Setting up a new VM is manual, time-consuming labor, and each time it is repeated, the output will be slightly different.

*Corresponding author.

johannes.witzig@zhaw.ch (J. Witzig); flavio.felder@zhaw.ch (F. Felder); vixay.phimmasane@zhaw.ch (V. Phimmasane); simon.kuenzli@zhaw.ch (S. Künzli); andreas.ruest@zhaw.ch (A. Rüst)

0009-0003-9311-0128 (J. Witzig); 0009-0000-7489-6354 (V. Phimmasane); 0009-0009-7448-7695 (S. Künzli)

The student's feedback for the VM solution revealed that the large size of the VM was especially problematic. Some students first had to free up space on their storage drives prior to the installation. Furthermore, VirtualBox's integrated file sharing feature proved to be unreliable.

2. Novelty of This Work

When using Zephyr in lab exercises, we need a way for the students to create, compile, and flash Zephyr applications. As time in the laboratories is limited, typically two hours per topic, we want the students to focus on the task instead of setting up their development environment. In other words, we try to maximize the time students spend on the laboratory's content while minimizing the time spent debugging the development environment. Zephyr is new for the students; every student has a different setup on their laptop. Additionally, the laboratory supervisors must be able to provide help efficiently. We therefore seek an environment that has the following properties.

- Uniform
- Reproducible
- Easy to set up and use
- Easy to maintain
- Works on Windows, macOS, and Linux (`aarch64` and `x86_64`)

The solution that we have now arrived at is based on a container. Students who use macOS or a Linux distribution can run the container using a container engine like *Podman* [3], while Windows users can use the container image as a WSL2 distro [4]. Compared to the VM, the container has the following notable advantages.

- Sufficiently reproducible
- Lean resource usage
- Small ($\approx$ 2 GiB / $\approx$ 0.5 GiB compressed)[1]
- Reliable file sharing
- Students can use their familiar editor to work on the assignments

We are aware that there is an official Zephyr developer container image [5]. However, this image contains things we do not need and is therefore a lot larger ($\approx$ 20 GiB).

One inconvenience that we still face is that WSL2 does not provide access to USB devices by default. This, for example, prevents us from flashing the firmware to the target from within the environment. Microsoft's solution to this problem is to use a tool that tunnels USB over IP [6]. We tested it on a few systems and ultimately decided not to use it. It was difficult to get the USB devices to work on some systems due to firewall issues. Instead, we now use SEGGER tools on the host operating system as an alternative. This works well as the compiled Zephyr binary inside the WSL2 distribution is easily accessible for tools that run on the host (Figure 1). Instead of programming the target using `west flash`, for example, we use the *J-Flash Lite* GUI tool on the host.

[1] The size of the container heavily depends on included zephyr modules.

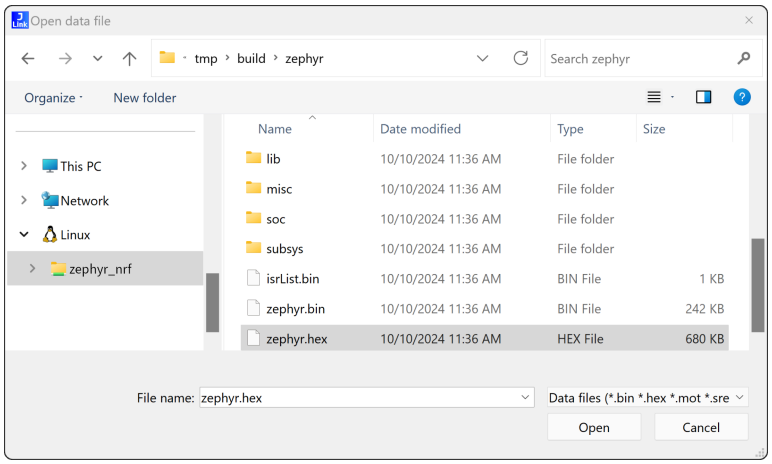

Figure 1: *Open data file* dialog from *J-Flash Lite* showing the Zephyr firmware binary inside the WSL2 distribution.

3. Implementation

Containers are built from a `Containerfile`. We use the latest stable Debian version as a base and install all required packages, the Zephyr SDK, and the Zephyr source code.

Our work is publicly available at [7]. The repository also includes the instructions that we hand out to the students.

The following is a simplified version of the `Containerfile`[1] that we use to build our container.

```
FROM debian:12-slim

# Install apt packages
# Install pip packages
# Install Zephyr SDK
# Clone Zephyr

WORKDIR /root/dev
CMD ["bash"]
```

A container image can then be built from this file using *Podman*[2]. Several build arguments are supported:

- The Zephyr version
- The SDK version
- Which Zephyr modules to install (all modules are installed when this is not specified)
- Which additional Python and system packages to install

The following command builds a container image with Zephyr v4.2.0 for Nordic Semiconductor's nRF platform.

```
podman build \
  --build-arg="ZEPHYR_VERSION=4.2.0" \
  --build-arg="SDK_VERSION=0.17.2" \
  --build-arg="MODULES=cmsis_6 hal_nordic mbedtls mcuboot segger" \
  . -t zephyr_nrf_v4.2.0
```

[1] The complete `Containerfile` can be found in Appendix A.

[2] *Podman* is an application for managing *Open Container Initiative* (OCI) containers. *Podman*'s command line interface is designed to be compatible with *Docker*'s.

Depending on whether the students want to run the container in a container engine or on WSL2, the following steps differ.

3.1. Container Engine

In order to provide the container to students who use a container engine, the container can be exported (and compressed) to a file with the following command.

```
podman save zephyr_nrf_v4.2.0 | zstd -19 > zephyr_nrf_v4.2.0.tar.zst
```

The students can then import the container using *Podman*'s `load` command.

```
podman load --input zephyr_nrf_v4.2.0.tar.zst
```

3.2. WSL2

When using Windows, the container can be used as a WSL2 distro by importing the container's filesystem. Microsoft calls this a *custom distro* [8]. The following commands export the container's filesystem.

```
cid=$(podman create zephyr_nrf_v4.2.0)
podman export $cid | zstd -T0 -19 > zephyr_nrf_v4.2.0_wsl.tar.zst
podman rm $cid
```

On Windows, the students can then import the filesystem as a WSL2 distro by running the following commands in *PowerShell*.

```
mkdir "C:\WSL\zephyr_nrf_v4.2.0"
wsl --import zephyr_nrf_v4.2.0 "C:\WSL\zephyr_nrf_v4.2.0" `
  zephyr_nrf_v4.2.0_wsl.tar.zst
wsl -d zephyr_nrf_v4.2.0
```

4. Results and Discussion

We currently use the containerized environment in three separate courses:

- Microcomputer Systems (BSc program)
- Embedded Real-Time Software (MSc program)
- Security in Embedded Systems (continuing education)

In the *Microcomputer Systems* course and the *Embedded Real-Time Software* course, we use Zephyr to study real-world implementations of general operating system concepts. This includes threads, scheduling (priorities, cooperative vs preemptive, time slicing), and resource locks (mutexes, semaphores). We also look at different debugging methods with SEGGER Ozone and SEGGER SystemView (Figure 3). For these courses, we use a development kit with a custom carrier board (Figure 2) that adds additional peripherals and allows the design of intuitive lab exercises. With approximately 10 hours of Zephyr-based lab exercises per course, we believe that the use of the proposed container solution outweighs a native Zephyr installation, despite its described limitations. The container solution's simplicity in terms of installation is especially appealing, considering the students' heterogeneous laptop setups.

Figure 2 shows the development kit in action, displaying the current output of a lab exercise featuring the dining philosophers problem that is used to teach the use of semaphores and mutexes with Zephyr. Figure 3 shows the respective events recorded by SEGGER SystemView.

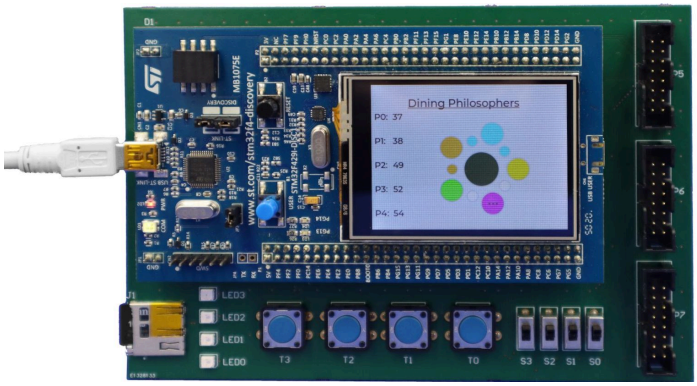

Figure 2: STM32F429I-DISC1 development kit attached to a custom carrier board running a dining philosophers application.

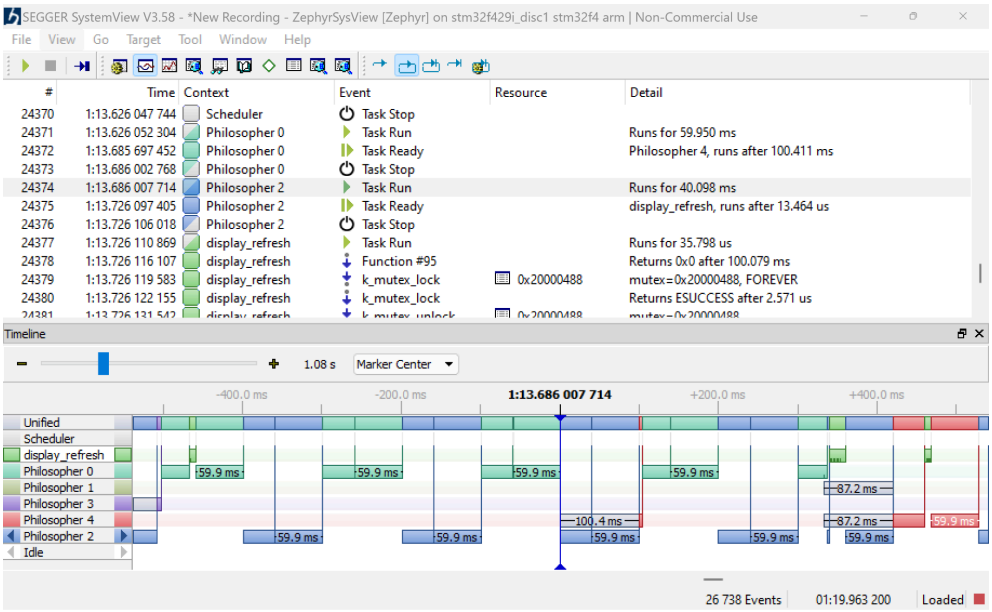

Figure 3: SEGGER SystemView showing events from the dining philosophers application.

Finally, in the *Security in Embedded Systems* course offered in continuing education, the participants study firmware security. The participants work on a proof-of-concept application that implements secure boot and secure over-the-air firmware updates using MCUBoot and Sysbuild. In another lab, we cover certificate-based authentication to a server using a secure element. In this course, we use a development kit with a secure element extension board (Figure 4).

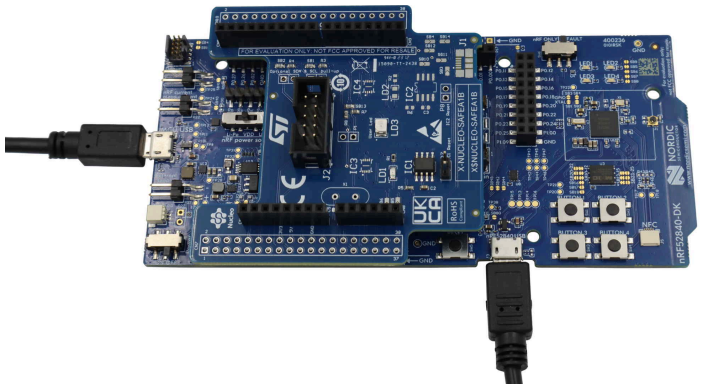

Figure 4: nRF52840 development kit with a secure element expansion board (X-NUCLEO-SAFEA1) attached.

5. Future Work

On the one hand, we are still gaining experience with the current container solution and hardware boards. On the other hand, we are currently working on replacing one of the boards (Figure 2) with a new, adapted custom board that uses an STM32H5 chip. Because of a design constraint (multiple use of data connections) in today's custom carrier board (see Figure 2), the display does not work with Zephyr versions newer than 3.1. Besides fixing this annoyance, the new chip that we plan to use has a Cortex-M33, which allows us to extend the security-related lab exercises on all course levels with ARM TrustZone.

Furthermore, we intend to gradually replace most of our course content with Zephyr-based versions. For instance, we are currently working on lab exercises for running tests on the target using Ztest and Twister.

6. Conclusion

As Zephyr sees wider adoption in our university, providing a uniform development environment for our students is important. Discussions with industry partners in our projects also suggest that Zephyr is becoming increasingly widespread in the industry.

We have tried several approaches to provide a uniform Zephyr development environment for our students. While a standard virtual machine has problems with reproducibility and uses a lot of resources, the container compensates for these disadvantages.

We have successfully used a container as a Zephyr development environment in several courses at our university, and it has proven its practicality in over 2 years of coursework.

The container is small, sufficiently reproducible, and easy to set up and maintain. It allows students to use their familiar editor while providing a uniform development experience, and allows both students and lecturers to focus on the lab tasks.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] M. Noseda and S. Künzli, "Attacking Secure-Element-Hardened MCUboot Using a Low-Cost Fault Injection Toolkit," in *Innovative Security Solutions for Information Technology and Communications*, Springer, 2024, pp. 126–143. doi: 10.1007/978-3-031-52947-4_10.

[2] F. Schranz, V. Phimmasane, A. Witzig, A. Rüst, and D. Schmid, "User Assistance System for Smart Commercial Buildings - Use Cases and Proof of Concept," in *Proceedings of the 2024 European Conference on Computing in Construction*, in Computing in Construction, vol. 5. Chania, Greece: European Council on Computing in Construction, July 2024. doi: 10.35490/EC3.2024.190.

[3] Podman Developers, "What is Podman?." Accessed: July 10, 2025. [Online]. Available: https://docs.podman.io/en/latest/

[4] Mirosoft Corporation, "What is the Windows Subsystem for Linux?." Accessed: July 10, 2025. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/use-custom-distro

[5] Zephyr Developers, "Zephyr Docker Images." Accessed: July 08, 2025. [Online]. Available: https://github.com/zephyrproject-rtos/docker-image

[6] Mirosoft Corporation, "Connect USB devices." Accessed: June 18, 2025. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/connect-usb

[7] Institute of Embedded Systems, Zurich University of Applied Sciences, "Zephyr Development Environment Container." Accessed: July 16, 2025. [Online]. Available: https://github.com/InES-IoT/zephyr_development_environment_container

[8] Mirosoft Corporation, "Import any Linux distribution to use with WSL." Accessed: June 18, 2025. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/use-custom-distro

A. Containerfile

```
FROM debian:12-slim

# Set non-interactive frontend for apt-get to skip any user confirmations
ENV DEBIAN_FRONTEND=noninteractive

# Install base packages
ARG ADDITIONAL_APT_PACKAGES=""
RUN apt-get -y update && \
  apt-get -y upgrade && \
  apt-get install --no-install-recommends -y \
  python3-pip \
  git \
  cmake \
  device-tree-compiler \
  wget \
  xz-utils \
  ninja-build \
  ${ADDITIONAL_APT_PACKAGES}

# Install Zephyr SDK
ARG SDK_VERSION
RUN test -n "${SDK_VERSION}" || (echo "SDK_VERSION build argument not set" && false)
WORKDIR /opt/toolchains
RUN HOST_ARCHITECTURE=$(uname -m) && \
  wget --quiet -O sdk.tar.xz https://github.com/zephyrproject-rtos/sdk-ng/releases/download/\
v"${SDK_VERSION}"/zephyr-sdk-"${SDK_VERSION}"_linux-"${HOST_ARCHITECTURE}"_minimal.tar.xz && \
  tar -xf sdk.tar.xz && \
  rm sdk.tar.xz && \
  cd zephyr-sdk-"${SDK_VERSION}" && \
  wget --quiet -O toolchain.tar.xz https://github.com/zephyrproject-rtos/sdk-ng/releases/download/\
v"${SDK_VERSION}"/toolchain_linux-"${HOST_ARCHITECTURE}"_arm-zephyr-eabi.tar.xz && \
  tar -xf toolchain.tar.xz && \
  ./setup.sh -t arm-zephyr-eabi -c && \
  rm toolchain.tar.xz zephyr-*.sh

# Clean up stale packages
RUN apt-get clean -y && \
  apt-get autoremove --purge -y && \
  rm -rf /var/lib/apt/lists/*

# Install python dependencies
ARG ADDITIONAL_PYTHON_PACKAGES=""
RUN pip install --break-system-packages west pyelftools pylink-square ${ADDITIONAL_PYTHON_PACKAGES} && \
  echo "export PATH=~/.local/bin:\"$PATH\"" >> ~/.bashrc

# Init the Zephyr workspace
ARG ZEPHYR_VERSION
RUN test -n "${ZEPHYR_VERSION}" || (echo "ZEPHYR_VERSION build argument not set" && false)
ARG MODULES=""
WORKDIR /root/zephyrproject
RUN git clone --branch v${ZEPHYR_VERSION} --depth=1 \
  https://github.com/zephyrproject-rtos/zephyr && \
  west init --local zephyr && \
  west update --narrow --fetch-opt=--depth=1 ${MODULES} && \
  echo "source \"$(pwd)/zephyr/zephyr-env.sh\"" >> ~/.bashrc

WORKDIR /root/dev
CMD ["bash"]
```