



Embedded Rust for C Programmers Lab Manual

BENINGO
EMBEDDED GROUP

Jacob Beningo

jacob@beningo.com

Social Links:



Table of Contents

Contents

Overview 3

The Instructor – Jacob Beningo..... 4

Recommended Materials..... 5

 Development Board(s) 5

 Software Tool(s) 5

 Visual Studio Code Plug-ins 6

Recommendations for Completing the Labs 6

Lab Exercises..... 8

 Lab Exercise 0 – Toolchain Installation..... 8

 Lab Exercise 1 – Containerize your Rust Environment..... 9

 Lab Exercise 2 – Building a Rust PAC and Hello Blinky Application..... 13

Overview

The C programming language has been a staple of embedded software development for 50 years. Many languages like Ada, C++, and others have attempted to usurp it, only to fail. Rust is a memory-safe systems programming language that has been gaining interest across various industries. This workshop will introduce the Rust programming language for experienced C programmers.

The focus will be on highlighting the similarities and differences between the two languages, emphasizing how Rust can provide improved memory safety and performance without sacrificing the low-level control that C programmers are accustomed to.

Attendees will learn the basics of Rust's syntax and standard library and best practices for writing safe and efficient code in Rust. By the end of the workshop, participants will have a solid understanding of Rust and will be able to start using it in their own projects.

Example topics covered in this workshop include:

- Similarities and differences between C and Rust
- An introduction to the Rust toolchain
- Memory mapped I/O
- How to utilize peripheral access crate (PAC) and HAL crate
- Best practices for developing embedded applications in Rust

I hope you enjoy the course and dramatically improve your understanding of embedded software development using Rust.

I hope you enjoy the course and dramatically improve your understanding of embedded software development.

If you have questions, run into issues, or have topics you'd like to hear more about or feedback on, please feel free to email jacob@beningo.com.

The Instructor – Jacob Beningo

Jacob Beningo, CSDP. Jacob Beningo is an embedded software consultant who dramatically transforms businesses by improving software quality, cost, and time to market. He regularly publishes articles, blogs, webinars, and white papers about software architecture, processes, and development skills on over half a dozen platforms. He has completed projects across several industries, including automotive, defense, medical, and space. Jacob has worked with companies in over a dozen countries and has software operating on Earth, in orbit, around the moon, on the moon, and someday around Mars. Jacob holds bachelor's degrees in electrical engineering, Physics, and Mathematics from Central Michigan University and a master's degree in Space Systems Engineering from the University of Michigan.



Contact Jacob at jacob@beningo.com

www.beningo.com contains additional resources, templates, and Jacob's monthly Embedded Bytes newsletter. Check out his workshops at <https://beningo.mykajabi.com/>

Click the social media link below to follow Jacob and get more tips and tricks:



The following sites contain Jacobs's blogs and materials. Access the materials by clicking the image:

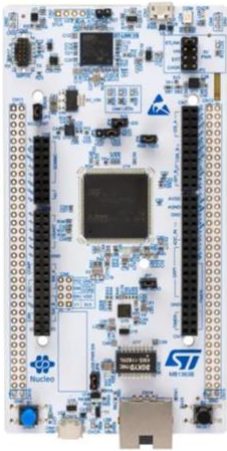


Recommended Materials

Development Board(s)

The workshop materials are designed to be used with the STM32U575 Nucleo Board.

Due to bandwidth limitations, Jacob won't be able to help support getting lab materials to work on any additional boards.

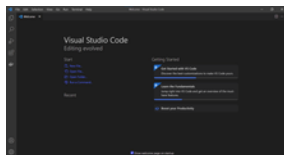


The STM32U575 Nucleo Board (NUCLEO-U575ZI-Q) is recommended for this course. It is a low-cost development platform from ST Microelectronics that includes a configuration tool for FreeRTOS, drivers, and frameworks to get examples up and running quickly. The board also has a built-in debugger, an Arduino expansion header, and several LEDs. It uses an Arm Cortex-M33 processor.

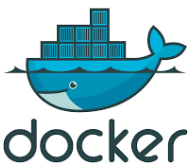
These boards are recommended for this course because they include a range of sensors perfect for interfacing when designing and building an RTOS-based application. Using an alternative board is possible but will require considerable rework.

Using an alternative board is possible but will require considerable rework.

Software Tool(s)



Visual Studio Code, also commonly called VS Code, is a source-code editor made by Microsoft with the Electron Framework for Windows, Linux, and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. It can be downloaded [here](#).



Docker makes development efficient and predictable. It eliminates repetitive, mundane configuration tasks and is used throughout the development lifecycle for fast, easy, and portable application development—desktop and cloud. Docker's comprehensive end-to-end platform includes UIs, CLIs, APIs, and security that are engineered to work together across the entire application delivery lifecycle. It can be downloaded [here](#).

One last tool that you will need to install if you are using Windows is make. If you don't want to, you can copy and paste any commands we are using in our makefile and execute them manually. You can use various tools like WSL2, Cygwin, or MinGW. I recommend using Chocolatey.

To install make, you can use Chocolatey by executing the following command in your Power Shell in admin mode:

```
Set-ExecutionPolicy Bypass -Scope Process -Force;  
[System.Net.ServicePointManager]::SecurityProtocol =  
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object  
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

If you ever need to upgrade, you can use the following command:

```
choco upgrade chocolatey
```

To test that the install was successful, run the command:

```
make -version
```

If successful, you should see version information for Make.

Visual Studio Code Plug-ins

There are several Visual Studio Code extensions that you will want to install to make the labs easier. The following are the recommended extensions that can be installed through the extensions menu:

- Cortex Debug
- Rust Analyzer

Recommendations for Completing the Labs

There are several options for how the attendee can complete each lab.

First, the most recommended method is to follow all the steps in the lab from scratch. Performing each lab from scratch will maximize your experience with the course material and topics. However, doing the labs from scratch is also the most time-consuming. This method may not allow you to complete the labs in the given timeframe in a time-constrained environment, such as during live on-site training.

Second, you can copy and paste the solution modules code for each step into your code project. This will decrease the time and potential error of writing all the lines of code yourself and reduce the time it takes to complete the labs.

Finally, you can import the lab solutions and play with them to understand the main concepts. This is the quickest way to complete the labs and may provide the most negligible value. The general way to master a technique or topic is to attend the lecture and hear about it, go through the lab hands-on to try it, and then implement that technique into your software.

Lab Exercises

Lab Exercise 0 – Toolchain Installation

Notes: Ensure you have watched and/or attended the Session 1 presentation.

Overview:


In this lab, developers will set up their computers with the tools necessary to perform the labs. Most toolchains support Windows, Linux, and macOS, but some tools may be supported on a single platform. When this occurs, I will try to note this. If the tool is not supported, you will be unable to perform that lab part. This should have minimal impact on your learning experience.

Lab Instructions:

This lab's purpose is to set up and configure the toolchains necessary to run the remaining labs successfully. Let's get started.

Steps:

Install Visual Studio Code

- 1) Visit <https://code.visualstudio.com/download>. Download Visual Studio Code for your operating system.
- 2) Open Visual Studio Code and click on the extensions (The  icon on the left).
- 3) There are several extensions that are useful for embedded development. One by one, search for and install the following extensions:
 - a. Cortex-Debug
 - b. C/C++ from Microsoft
 - c. Embedded Tools
 - d. Rust Analyzer

Install Docker

- 4) Visit <https://www.docker.com/products/docker-desktop/>. Download Docker and follow the installation instructions for your operating system.

Congratulations! You have completed the toolchain setup(s) and are now ready for the next lecture and lab.

Lab Exercise 1 – Containerize your Rust Environment

Notes: Ensure you have watched and/or attended the Session 1 presentation.

Overview:

In this lab, you will set up the Rust environment on your machine inside of a Docker container.

Lab Instructions:

For this lab, you'll need your PC and an internet connection. You won't need your development board quite yet. The goal is to have a containerized development environment that you can use to build Rust applications.

The major steps that you will take to accomplish this includes:

- 1) Creating a Docker Container
- 2) Verify your Docker Container with Rust
- 3) Create a "Hello World!" application
- 4) Modify "Hello World!" to become familiar with Rust functions

Steps:

Create a Docker Container

The first goal of this lab is to create a Dockerfile that outlines the instructions for creating a container. All the instructions will install the tools necessary for embedded Rust development.

- 1) Copy the 00-Starters/LAB_1 folder to a directory from which you'd like to work on the labs.
- 2) Open Visual Studio Code. From the File menu, select Open Folder and navigate to your svd2rustU5 folder in the LAB_1 directory.
- 3) You'll notice that there is a Dockerfile within the docker folder along with a project.sh script.
- 4) Open the Dockerfile
- 5) On line 29, you'll find a TODO comment. Go to the Rust website and find the install instructions for Rust on Linux.

Note: I'd also recommend you look at what tools we are installing.

- 6) Next, you need to choose the hardware architecture for your chip. We'll be using the Cortex-M33. Find the architecture type and uncomment the correct architecture.
- 7) In the project.sh file, replace the text YOURNAME with your last name or your Docker username.
- 8) You are now ready to build your container. If you haven't already done so, start Docker.
- 9) Now in a terminal, run project.sh to build your image by typing:

```
./project.sh docker_image
```

Note: If you are using Windows, you'll have to manually type the command:

- 10) `docker build -t YOURNAME/embedded-rust-dev -f docker/Dockerfile .`

Note: Depending on your computer and internet connection, the container can take 2 – 10 minutes to build.

Verifying your Docker Container with Rust

Now that you have a containerized Rust build environment, we can verify that our Rust environment is set up correctly.

- 11) You can run the script devManager.sh as follows to start docker:

```
./devManager.sh docker_run
```

Note: If you are on windows, you'll need to use:

```
docker run --rm -it --privileged -v "%CD%:/home/app" YOURNAME/embedded-rust-dev:latest
```

- 12) A simple check to ensure that Rust is set up is to run the following command:

```
rustup --version
```

You should find that it returns something like:

```
root@8ba02db1be40:/home/app# rustup --version
rustup 1.27.1 (54dd3d00f 2024-04-24)
info: This is the version for the rustup toolchain manager, not the rustc compiler.
info: The currently active `rustc` version is `rustc 1.79.0 (129f3b996 2024-06-10)`
root@8ba02db1be40:/home/app#
```

The version information tells us that the Rust environment is set up within our container and is now ready for use. Let's create a quick "Hello World!" application.

Hello World!

You will now use the cargo package manager to create a simple "Hello World" application that you'll compile and run using your containerized environment.

- 13) In your terminal running Docker, create a new `hello_world` application using the following command:

```
cargo new hello_world
```

You should find that you now have a `hello_world` file in your project directory.

- 14) Navigate into the directory in the terminal using "`cd hello_world`".
- 15) Take a moment in Visual Studio Code to review the files that have been created. You'll notice files such as:
- a. `main.rs`
 - b. `Cargo.toml`
- 16) Open `main.rs`
- 17) Review the file and the syntax. Make sure that you understand what the application is doing.
- 18) In the terminal, compile the Rust application using:

```
cargo build
```

- 19) Finally, run the application:

```
cargo run
```

You should see "Hello, world!" in your terminal.

Playing with Rust Functions

Now that you have a working Rust application let's modify the application to use function calls so you get some experience with Rust syntax.

20) Change the main function to print the results returned from a function named hello:

```
println!("{}", hello());
```

21) Write the hello function so that it returns a string slice and takes a bool variable x as a parameter.

22) In the hello function, create a conditional statement to check whether x is true. If it is, using an explicit return statement to return “Hello World”. If it is not, then return it at the end of the function implicitly.

23) Compile and run the program to see “Hello World!”.

24) Now, in your if statement, remove one of the = signs from ==.

25) Compile the application. Notice how the Rust compiler catches that you are assigning a value to x, not checking for equality!

26) Fix the compilation error.

Congratulations! You now have some experience writing a basic Rust application!

If you’d like to go further with Rust syntax, I’d recommend that you work through [100 Exercises To Learn Rust](#). (It’s not embedded, but it covers the syntax well!).

Lab Exercise 2 – Building a Rust PAC and Hello Blinky Application

Notes: Ensure you have watched and/or attended the Session 1 presentation.

Overview:

In this lab, you will develop a blinky LED application for the STM32U575 Nucleo board. The lab will require you to develop your own peripheral access crate (PAC). You'll also need to design and implement a blinky application in the Rust language.

Lab Instructions:

For this lab, there are several major steps that you will need to work through to blink an LED on the development board. These steps include:

- 1) Creating the STM32U575 PAC
- 2) Creating a blinky application from an embedded application template
- 3) Updating the template for the STM32U575
- 4) Writing the blinky application
- 5) Configuring debug scripts and launch configurations

Let's get started!

Steps:

Creating the STM32U575 PAC

Before you can access low-level registers on the microcontroller, you need a crate that exposes the functions of the board. To do this, we'll use `svd2rust` to create the PAC.

- 1) Copy the LAB_2 starter project to your preferred location.
- 2) Open the folder in Visual Studio Code along with opening a new terminal.
- 3) Start your Docker container by running `./devManager.sh docker_run`. You should be in the root directory of the LAB_2 folder. (Although it will be `/home/app` in the container. You can use `ls` to verify the files match what you expect).
- 4) In your Docker container, create a new library project using the following command:

```
cargo new stm32u575_pac --lib
```

- 5) In your terminal, navigate to the `stm32u575_pac` folder.
- 6) You'll notice in the starter project that there is a `svd` folder. Copy its contents to the root of the new project. These are the files we will process to create the PAC.

Note: The svd files from ST do not process well. The Rust community has a project that patches them so that the svd files make sense. We will use the patched version. I've included the original in case you want to see the differences between the files.

- 7) Run the following command to convert the svd file to rust crates.

```
svd2rust -i stm32u575.svd.patched
```

The resulting output should look something like the following:

```
root@ba00886dd06e:/home/app/stm32u575_pac# svd2rust -i stm32u575.svd.patched
[INFO svd2rust] Parsing device from SVD file
[INFO svd2rust] Rendering device
[WARN svd2rust::generate::register] Missing description for register PWR_SVMSR
[WARN svd2rust::generate::register] Missing description for register PWR_SVMSR
[WARN svd2rust::generate::register] Missing description for register SPI_CR1
[WARN svd2rust::generate::register] Missing description for register SPI_CR1
[WARN svd2rust::generate::register] Missing description for register SPI_CR2
[WARN svd2rust::generate::register] Missing description for register SPI_CR2
[WARN svd2rust::generate::register] Missing description for register SPI_IER
[WARN svd2rust::generate::register] Missing description for register SPI_IER
[WARN svd2rust::generate::register] Missing description for register SPI_SR
[WARN svd2rust::generate::register] Missing description for register SPI_SR
[WARN svd2rust::generate::register] Missing description for register SPI_IFCR
[WARN svd2rust::generate::register] Missing description for register SPI_IFCR
[WARN svd2rust::generate::register] Missing description for register SPI_AUTOCR
[WARN svd2rust::generate::register] Missing description for register SPI_AUTOCR
[WARN svd2rust::generate::register] Missing description for register SPI_TXDR
[WARN svd2rust::generate::register] Missing description for register SPI_TXDR
[WARN svd2rust::generate::register] Missing description for register SPI_RXDR
[WARN svd2rust::generate::register] Missing description for register SPI_RXDR
[WARN svd2rust::generate::register] Missing description for register SPI_TXCRC
[WARN svd2rust::generate::register] Missing description for register SPI_TXCRC
[WARN svd2rust::generate::register] Missing description for register SPI_RXCRC
[WARN svd2rust::generate::register] Missing description for register SPI_RXCRC
[WARN svd2rust::generate::register] Missing description for register UCPD_RX_ORDSETR
[WARN svd2rust::generate::register] Missing description for register UCPD_RX_ORDSETR
[WARN svd2rust::generate::register] Missing description for register UCPD_RX_PAYSZR
[WARN svd2rust::generate::register] Missing description for register UCPD_RX_PAYSZR
[WARN svd2rust::generate::register] Missing description for register UCPD_RXDR
[WARN svd2rust::generate::register] Missing description for register UCPD_RXDR
```

Note: The warnings are okay. They just tell us that a description is missing. (I've notified ST about the issue.)

- 8) If you look at the src folder, it just contains some default crate. Delete it using the following terminal command:

```
rm -rf src
```

- 9) When you ran svd2rust, it created just a single lib.rs crate. Every peripheral and definition is in just a single crate! That's not very modular. Let's parse that library.rs file using the form application with:

```
form -i lib.rs -o src/ && rm lib.rs
```

This will parse lib.rs and remove it once we are done. You can review the src folder to see everything that was created. In fact, it would be a good idea to poke around a few of the modules, too!

- 10) You probably noticed that the code is not very readable. Let's format the code using the following command:

```
cargo fmt
```

- 11) Before compiling the code, we need to adjust the dependencies. Open the Cargo.toml file.

- 12) Update it so that it has the following dependencies:

```
[package]
name = "stm32U575_pac"
version = "0.1.0"
edition = "2021"

[dependencies]
critical-section = { version = "1.1.2", optional = true }
cortex-m = { version = "0.7.7" }
cortex-m-rt = { version = "0.7.3" }
vcell = "0.1.3"

[features]
rt = ["cortex-m-rt/device"]
```

Note: See the lecture slides for what all these things mean.

- 13) Now compile the PAC using the following cargo command:

```
cargo build -r
```

Congrats! You’ve successfully created your PAC! We can use it inside our blinky application to access low-level peripherals and registers!

Creating Hello Blinky from a Template

The easiest way to get an Embedded Rust application up and running is to use a template and then modify it. That’s exactly what we are going to do.

- 14) In your terminal, make sure that you are in the root directory of the LAB_2 folder. Once you have verified this, use cargo generate to create:

```
cargo generate --git https://github.com/rust-embedded/cortex-m-quickstart
```

Note: You might get an error like:

Error: could not determine the current user, please set \$USER

If this happens, just run the following command and then repeat the cargo generate command:

```
export USER=root
```

- 15) You should now be queried for a project name. Use the name “blinky”. It will take a moment, but the example template will create a new project.
- 16) Using Visual Studio Code, open the blinky folder.
- 17) Before you examine what is there, open a new terminal and launch your Docker container again.
- 18) Take a moment to explore the project directory and files. You’ll notice the following:
- a. main.rs with a starter application
 - b. An example folder with examples based on common embedded needs
 - c. A .vscode folder with configurations
 - d. A memory.x linker file
 - e. OpenOCD configurations

The template is a great baseline, but right now it’s configured to use the STM32F3 Discovery board! That’s not the board we have. Let’s make some updates.

Updating the Template for the STM32U575 Nucleo Board

The template requires a few modifications for us to use the STM32U575. We’ll now make those adjustments.

- 19) Open `memory.x`.
- 20) You'll notice that it contains a few memory descriptions. These descriptions are not correct for the STM32U575. Update them to match the following:

```
MEMORY
{
    /* NOTE 1 K = 1 KiBi = 1024 bytes */
    FLASH : ORIGIN = 0x08000000, LENGTH = 2048K
    RAM : ORIGIN = 0x20000000, LENGTH = 768K
}
```

- 21) Now, we need to make sure that the build environment understands what core we are building for. Open the `config.toml` file located in the `.cargo` folder.

Note: `config.toml` contains configurations for running the application and building for our target.

- 22) Comment out the `"thumbv7m-none-eabi"` target on line 32.
- 23) Uncomment out the `"thumbv8m.main-none-eabihf"` target on line 37. Our processor is a Cortex-M33 with FPU.
- 24) Save the file and close it.
- 25) In your terminal, where you should be in your Docker environment within the `blinky` folder, run the following command to add support for `"thumbv8m.main-none-eabihf"`:

```
rustup target add thumbv8m.main-none-eabihf
```

- 26) Open `Cargo.toml`.

Notice that it has a whole bunch of dependencies and application settings.

- 27) You might have noticed that the versions for `cortex-m` and `cortex-m-rt` are not the same that you used in the PAC. Update these to the latest crate versions:

```
cortex-m = { version = "0.7.7", features = ["critical-section-single-core"] }
cortex-m-rt = "0.7.3"
```

- 28) One last change to the dependencies is that you want to use the PAC you created for the STM32U575. You can add this dependency by adding the following line under the `[dependencies]` section:

```
stm32u575_pac = { path = "../stm32u575_pac", features = ["rt", "critical-section"] }
```

Writing the Blinky Application

The development board that we are using has three LEDs: red, green, and blue. We are going to write an application that blinks the green LED.

29) Open main.rs

30) You'll want to use the pac we just created. To do this, in the use section, add the following line of code:

```
use stm32u575_pac as pac;
```

31) You'll also want to add the ability to access the cortex-m cpu peripherals by adding the following line of code:

```
use cortex_m::peripheral::Peripherals;
```

32) In main remove the nop operation, which is no longer needed. You'll also want to remove the following line of code so that you don't get any compiler warnings:

```
use cortex-m::asm;
```

33) There are two things you need to do in main before getting to the loop:

- a. Get the device peripheral access
- b. Get the cpu peripheral access

This can be done by creating two variables, dp and cp as follows:

```
let dp = pac::Peripherals::take().unwrap();  
let mut cp = Peripherals::take().unwrap();
```

34) Before we can blink an LED, we must enable the GPIO blocks we want to use. The STM32U575 has LEDs on the following ports and pins:

```
LD1: PC7  
LD2: PB7  
LD3: PG2
```

35) The enable bits are in the rcc peripheral. Add a line of code to get access to the rcc by doing the following:

```
let rcc = dp.rcc;
```

36) Now we can enable the ports by accessing the PAC in the following way:

```
// ahb2enr1: AHB2 peripheral clock enable register 1
// bit1: GPIOBEN: IO port B clock enable
// bit2: GPIOCEN: IO port C clock enable
// bit6: GPIOGEN: IO port G clock enable
rcc.rcc_ahb2enr1().modify(|_, w| {
    w.gpioben().set_bit() // enable GPIOB (bit 1)
    .gpiocen().set_bit() // enable GPIOC (bit 2)
    .gpiogen().set_bit() // enable GPIOG (bit 6)
});
```

Please type this out yourself so that you can see the options that are available from the IntelliSense.

37) Create a couple variables to individually access the different gpio ports:

```
// Get the gpio peripherals needed to blink LEDs
let gpiob = dp.gpiob;
let gpiorc = dp.gpioc;
let gpiorg = dp.gpiog;
```

38) The next bit is slightly tricky. Because we are accessing low-level bits, the PAC has marked the next operations as unsafe. That means we must use the unsafe keyword for the accesses to prevent a compiler error. The code you want to use to set the port and pin as an output, instead of the default input, is below:

```
unsafe {
    // GPIO MODER: GPIO port mode register
    // GPIO modeX: Port x configuration bits (y = 0..15)
    // 00: Input mode (reset state)
    // 01: General purpose output mode
    // 10: Alternate function mode
    // 11: Analog mode

    // Set PB7 to output
    gpiob.gpio_moder().modify(|_, w| w.mode7().bits(0b01));

    // Set PC7 to output
    gpiorc.gpio_moder().modify(|_, w| w.mode7().bits(0b01));

    // Set PG2 to output
    gpiorg.gpio_moder().modify(|_, w| w.mode2().bits(0b01));
}
```

39) The loop code is just setting and clearing the output bit for the specific port and pin. For the green LED, this can be done with:

```
loop {  
    gpioc.gpio_odr().modify(|_, w| w.od7().set_bit());  
    delay(&mut cp.SYST, 160_000);  
  
    gpioc.gpio_odr().modify(|_, w| w.od7().clear_bit());  
    delay(&mut cp.SYST, 160_000);  
}
```

40) You may not realize this, but we need to implement that delay function. It's a quick and dirty delay. I've implemented it as the following:

```
fn delay(syst: &mut cortex_m::peripheral::SYST, ticks: u32) {  
    syst.set_reload(ticks);  
    syst.clear_current();  
    syst.enable_counter();  
  
    while !syst.has_wrapped() {}  
}
```

41) You now have an application that is ready to build and test. Let's get the build part done. In Docker, type cargo build. You should see the following if you've set everything up properly:

```
PROBLEMS 61 OUTPUT TERMINAL PORTS MEMORY XRTOS COMMENTS

Compiling stm32u575_pac v0.1.0 (/home/app/stm32u575_pac)
Compiling blinky v0.1.0 (/home/app/blinky)
Compiling panic-halt v0.2.0
Compiling nb v0.1.3
Compiling semver v0.9.0
Compiling volatile-register v0.2.2
Compiling embedded-hal v0.2.7
Compiling rustc_version v0.2.3
Compiling bare-metal v0.2.5
Compiling quote v1.0.36
Compiling cortex-m-rt-macros v0.7.0
Finished `dev` profile [unoptimized + debuginfo] target(s) in 45.23s
root@20d054703a23:/home/app/blinky#
```

You are now ready to configure your debug and launch scripts!

Configuring Debug and Launch Scripts

Now that you have an application that should blink an LED, we want to test it on hardware! That requires us to set up the debug configurations so that we can program and run the application on target.

- 42) Under the .vscode folder, open launch.json. You'll notice several configurations that range from running QEMU to debugging on target.
- 43) Modify the second configuration that has the type "cortex-debug". Update the following:
 - a. Executable should now be:

"./target/thumbv8m.main-none-eabihf/debug/blinky"

- b. Device should now be:

"STM32U575ZIT"

- c. Update configFiles so that the target is:

"target/stm32u5x.cfg"

- d. Finally, update the svdFile to:

"\${workspaceRoot}/.vscode/stmu575.svd"

- 44) Save launch.json.

- 45) In the LAB_2 solution, copy the stmu575.svd file and paste it into the .vscode folder.

- 46) In Visual Studio Code, on the left, click the debug tab. You'll see a green play button. In the dropdown next to it, select Debug OpenOCD.
- 47) Click the green play button. Your code might rebuild, but at the end, you should find yourself in your main.rs file inside the main function.
- 48) Click the run button and watch as your green LED blinks!

Congratulations! You've just written and deployed your first Embedded Rust application!

From here, you might try the following:

- Blink the other LEDs
- Create a blink pattern
- Create a tuple table with iterative loop to display an LED pattern
- Explore the other examples, such as semi-hosting and ITM