APPENDIX A: SECURITY TERMINOLOGY DEFINITIONS

Security is not the core focus of this book, but security is essential for many teams that want to develop embedded software. Throughout the book, there were several security-related terms mentioned that I was not able to go into deeper detail on. Appendix A provides a high-level definition for several of these terms.

DEFINITIONS

Access control – The device authenticates all actors (human or machine) attempting to access data assets. Access control prevents unauthorized access to data assets. Counters spoofing and malware threats where the attacker modifies firmware or installs an outdated flawed version.

Authenticity – Indicates that we can verify where and from whom the data asset came.

Arm TrustZone – Arm Cortex-M hardware technology for creating hardware-based isolation in microcontrollers. Cortex-A processors also have a version of TrustZone, but the implementation and details are different.

Attestation[1] – Attestation provides a mechanism for software to prove its identity to a remote party such as a server. Attestation data is signed by the device and verified using a public key by the request server.

Chain-of-trust – Established by validating each hardware and software component in a system from power-on through external communications. Each component's integrity and authenticity is checked before allowing it to operate.

Confidentiality – Indicates that an asset needs to be kept private or secret.

Denial of service – An interruption in an authorized user's access to a de-

vice or computer network caused by malicious intent.

Escalation of privilege[2] – A network attack that is used to gain unauthorized access to systems.

Firmware authenticity – The device verifies firmware authenticity before boot and upgrade. Counters malware threats.

Impersonation[3] – Is when a malicious actor pretends to be a legitimate user or service to gain access to protected information.

Integrity – Indicates that the data asset needs to remain whole or unchanged.

Malware[4] – A file or code, typically delivered over a network, that infects, explores, steals, or conducts virtually any behavior an attacker wants.

Man-in-the-middle[5] – A form of active wiretapping attack in which the attacker intercepts and selectively modifies communicated data to masquerade as one or more of the entities involved in a communication association.

MPU – Memory protection unit. Used to separate memory into memory regions and limit permissions for those regions.

NSPE – Nonsecure processing environment. An NSPE is the “feature-rich” application execution environment in a secure microcontroller-based system.

PPU – Peripheral protection unit.

PSA[6] – The Platform Security Architecture (PSA) is a family of hardware and firmware security specifications, as well as open source reference implementations, to help device makers and chip manufacturers build best-practice security into products.

PSA Certified[7] – A security certification scheme for Internet of Things (IoT) hardware, software, and devices. It was created by seven stake-

holder companies as part of a global partnership. The security scheme was created by Arm Holdings, Brightsight, CAICT, Prove & Run, Riscure, TrustCB, and UL.

Repudiation – An attack that can occur against the system or application in which the system does not have adequate controls in place to detect that the attack has occurred.

Root-of-Trust – This is an immutable process or identity which is used as the first entity in a trust chain. No ancestor entity can provide a trustable attestation (in Digest or other form) for the initial code and data state of the Root-of-Trust.

The Root-of-Trust often includes security functions such as initialization, software isolation, secure storage, firmware updates, secure state, cryptography functions, attestation, audit logs, and debug capabilities.

Secure communication – The device authenticates remote servers, provides confidentiality (as required), and maintains the integrity of exchanged data. Counters man-in-the-middle (MitM) threats.

Secure state – Ensures that the device maintains a secure state even in case of failure to verify firmware integrity and authenticity. Counters malware and tamper threats.

Secure storage – The device maintains confidentiality (as required) and integrity of data assets. Counters tamper threats.

SMPU – Shared memory protection unit.

SPE – Secure processing environment. A hardware isolated execution environment including memory and peripherals that contains the Root-of-Trust and secure application services.

SRAM PUFs[8] – SRAM physical unclonable function is a physical entity embodied in a physical structure. PUFs utilize deep submicron variations that occur naturally during semiconductor production and which give

each transistor slightly random electric properties – and therefore a unique identity.

Tamper[9] – An intentional but unauthorized act resulting in the modification of a system, components of systems, its intended behavior, or data.

APPENDIX B: 12 AGILE SOFTWARE PRINCIPLES

Over the past several decades, the agile movement has become a major methodology within the software industry. Unfortunately, there have been many offshoots where teams think they are "doing Agile" but are doing nothing of the sort. There are lots of mixed messages, and unfortunately the core message is often lost among the noise. I thought it would be helpful to remind developers of the 12 core agile principles that are outlined on the Agile Manifesto.

12 AGILE SOFTWARE PRINCIPLES[10]

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

APPENDIX C: HANDS-ON – CI/CD USING GITLAB

AN OVERVIEW

In Part 2, we discussed several modern processes involving DevOps and testing. Appendix C is designed to give you a little hands-on experience putting together a CI/CD pipeline to compile and test embedded software for an STM32 microcontroller. Appendix C will guide you on how to

- Set up Docker
- Configure an image for CI/CD
- Integrate code with a CI/CD pipeline
- Run unit tests
- Deploy built code to an STM32 development board

The example that we are about to walk through is exactly that. It is not meant for production code. The example will provide you with some hands-on experience and help to solidify the skills we've discussed throughout this book.

BUILDING STM32 MICROCONTROLLER CODE IN DOCKER

Docker provides a mechanism to set up our build, analysis, and testing environments in a container, making it easier to integrate into a CI/CD pipeline. Developers can set up a container whether they are running Windows, Linux, or macOS. One environment that can be used across multiple platforms. Containers also have the added benefit that new developers to the team don't need to spend a bunch of time setting up software and trying to duplicate the build environment. Instead, as part of the repo, the new developer only needs to run the image build commands, and the latest image with the code will be built and ready to go.

Before we walk through how to set up a Docker container for the STM32, I would recommend that you do the following:

- Install the latest version of **Docker**[11] and **Docker Compose**[12] on your development machine.
- Install the latest version of **STM32CubeIDE**.[13]
- Create an empty Git repo to save our project and files to.
- Find any STM32 development board that you have lying around the office (optional: only needed if you plan to deploy your code to a board).

Once you've prepared these items, you'll be ready to walk through setting up our build process within a Docker container.

INSTALLING GCC-ARM-NONE-EABI IN A DOCKER CONTAINER

The STM32 is an Arm Cortex-M–based 32-bit architecture. Typically, you would install your compiler as part of the IDE toolchain. However, we will install the GCC-arm-none-eabi compiler as part of a Docker image. For this example, I will use the October 2021 release of the GCC-arm-none-eabi compiler. I would recommend reviewing what the latest version is and using that one.

Our Dockerfile will have four sections:

1. The base image that we build on top of
2. Support tool installation
3. GCC-arm-none-eabi compiler installation (and configuring our path)
4. Setting up our working directory

The entire Dockerfile contents to create these sections and set up our image can be seen in Listing **C-1**. If you plan to follow along, I recommend creating the Dockerfile in the root of your Git repo and then pasting it into the code listing. In our setup script, you will notice that we are using Linux within our image. If you are running Windows, you don't need to worry about using Cygwin or any other Linux emulator. The Linux distribution is being run within the container that is running on top of Windows. We only need to worry about installing the tools we want within our image.

```
FROM ubuntu:latest
ENV REFRESHED_AT 2022-06-01
```

```
# Download Linux support tools
RUN apt-get update && \
    apt-get clean &&  \
    apt-get install -y \
        build-essential \
        wget \
        curl \
        git
# Set up a development tools directory
WORKDIR /home/dev
ADD . /home/dev
RUN wget -qO- https://developer.arm.com/-/media/Files
/downloads/gnu-rm/10.3-2021.10/gcc-arm-none-
eabi-10.3-2021.10-x86_64-linux.tar.bz2 | tar -xj
# Set up the compiler path
ENV PATH $PATH:/home/dev/gcc-arm-none-eabi-10.3-2021.10/bin
WORKDIR /home/app
```

Listing C-1 The complete Dockerfile source for creating a Docker image for the STM32 build process

Let's now break the Dockerfile down and examine what each section is doing and why.

The first line in the Dockerfile specifies the base image on which we will be basing our image. We will use the latest version of Ubuntu for our base image using the following Docker command:

```
FROM ubuntu: latest
```

> **Best Practice** It is possible that updates to the latest image could break your image. You may want to consider specifying a specific. For example, specify ubuntu 18.04 using `FROM ubuntu:18.04`.

Next, we need several support tools in Linux to download, unzip, install, and then configure the Arm compiler. We will install these tools in one RUN command as shown in Listing C-2. If additional tools were needed, we could continue to add the commands to install them to the bottom of this RUN statement.

```
# Download Linux support tools
RUN apt-get updated && \
    apt-get clean && \
    apt-get install -y \
       build-essential \
       wget \
       curl
```

Listing C-2 The Dockerfile command to download and install Linux support tools

Before we install GCC-arm-none-eabi, we want to specify where to install it within the Docker image. It is common to install it in a directory such as /home/dev. We can do this in our Dockerfile using the code shown in Listing C-3.

```
# Set up a development tools directory
WORKDIR /home/dev
ADD . /home/dev
```

Listing C-3 Specify where the tool directory is located

Now, we are ready to add the command to install the compiler using Listing C-4.

```
RUN wget -qO- https://developer.arm.com/-/media/Files
/downloads/gnu-rm/10.3-2021.10/gcc-arm-none-
eabi-10.3-2021.10-x86_64-linux.tar.bz2 | tar -xj
```

Listing C-4 The Dockerfile command to run to install the Arm compiler within the Docker container

> **Note** I am requesting a specific compiler version, 10.3-2021.10. You may want to update to the latest version when you read this. The latest version numbers can be found at **https://bit.ly/3LX8zqt**.

With the compiler command now included, we want to set up the PATH variable to include the path to the compiler. This will allow us to use make to compile our application. The command to add to the Dockerfile can be seen in Listing C-5.

```
# Set up the compiler path
ENV PATH $PATH:/home/dev/gcc-arm-none-eabi-10.3-2021.10/bin
```

Listing C-5 Set up the compiler path in the Dockerfile

Finally, we can set our working directory for our application code to the /home/app directory using

```
WORKDIR /home/app
```

We now have a Dockerfile that can be built to create our image. Our image can then be run to create our container where we have a virtual environment from which to build our code.

CREATING AND RUNNING THE ARM GCC DOCKER IMAGE

We now have a Dockerfile that contains all the commands necessary to create a Docker image, but the image does not exist yet. To create the image and then run it, there are a few commands that we need to run first. The first command is the Docker build command. The build command reads in our Dockerfile and then creates a Docker image based on the commands in the file. For example, to build the Dockerfile we just created, we would open a terminal in our Git repo root directory and type the following command:

```
docker build -t beningo/GCC-arm .
```

Notice that in the build command, we are using the -t option to tag our image.

In the example, we are tagging the image as beningo/GCC-arm. The resulting output in the terminal from the build command should look something like Figure C-1.

```
docker build -t beningo/gcc-arm .
[+] Building 54.6s (8/8) FINISHED
 => [internal] load build definition from Dockerfile                    0.0s
 => => transferring dockerfile: 734B                                    0.0s
 => [internal] load .dockerignore                                       0.0s
 => => transferring context: 2B                                         0.0s
 => [internal] load metadata for docker.io/library/ubuntu:latest        0.0s
 => CACHED [1/4] FROM docker.io/library/ubuntu:latest                   0.0s
 => [2/4] RUN apt-get update &&     apt-get clean &&      apt-get instal  24.6s
 => [3/4] RUN wget -qO- https://developer.arm.com/-/media/Files/download  26.0s
 => [4/4] WORKDIR /home/app                                             0.0s
 => exporting to image                                                  3.8s
 => => exporting layers                                                 3.8s
 => => writing image sha256:a651eee9482f512619ee78cda7d53c07bfc4dbf655fef  0.0s
 => => naming to docker.io/beningo/gcc-arm                              0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and l
earn how to fix them
beningo@Jacobs-MacBook-Pro stm32-example %
```

Figure C-1 The terminal output from running the Docker build command on our Dockerfile

The first time you build an image, it may take several minutes to create and run all the commands. For example, when I built the Dockerfile, it took 58.2 seconds the first time I built it. However, if you build it a second time, it will run much faster because only changes to the image will need to execute.

Once we have an image built, we are ready to run the image. The image can be started using the following command:

```
docker run --rm -it --privileged -v "$(PWD):/home/app"
beningo/gcc-arm:latest bash
```

The run command looks a bit complicated and overwhelming, but it's relatively straightforward. First, `--rm` is telling Docker that when we exit the container, we want to remove the image. Docker images usually require gigabytes of hard drive space, so we don't want to accumulate a whole bunch of them! Next, `-it` instructs Docker to allocate a pseudo-TTY connected to the container's stdin, creating an interactive bash shell in the container.[14] Next, the `--privileged` flag gives all capabilities to the container. Finally, the `-v` option mounts the current working directory into the container. This allows us to access the host file system, which is useful when we want to compile our code and easily access the generated object and binary files. All the stuff that comes after `-v` specifies where the

mount is located and what Docker image file tag we are running.

At this point, we now have a Docker image, a running container, and a command prompt that allows us to use the container we have running. The next thing we want to do is build something! However, to do so, we need a test project. So, let's now create an STM32 test project that we can build using our Docker container.

CREATING AN STM32 TEST PROJECT

To test our Docker build environment, there are several options available to us. First, we just create a few C modules and write a makefile. To test the environment, we don't necessarily need to build a microcontroller, but what's the fun in that? For this example, we will leverage the STM32 project generator tool STM32CubeMx. The STM32CubeMx tool will allow us to create a makefile-based project that we can put right into the root directory of our Git repo.

As I mentioned earlier, any STM32 board around your office will work. I have an STM32L475 IoT Discovery board that I use in my RTOS courses. So, for this example, that is what I'm going to use as my target. The exact target here isn't important; it's the general process that I want you to focus on. I recommend creating a new STM32CubeMx project named Blinky and initializing your development board to the default settings. I won't discuss how to do this because the toolchain walks you through it.

STM32CubeMx allows developers to configure development boards and STM32 microcontroller projects easily. Once the project is created, developers can use the project settings to choose their development environment. For example, developers can use a makefile, CubeIDE, Keil, and IAR Embedded Workbench. The trick is to set up the project to be makefile based. If you set the project up for a different toolchain, you can change it under the Project Manager in the Code Generation section shown in Figure C-2.

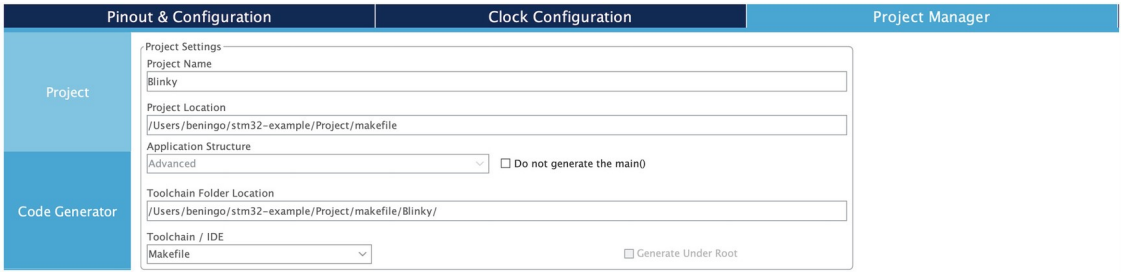

Figure C-2 The STM32CubeMx tool can be configured for several toolchains. For example, under Project Manager ➤ Code Generator, developers can set the Toolchain/IDE to Makefile

From this point, an example project can be created by clicking the generate code button. Once done, my project directory, the root of my repo, will look something like Figure C-3. Notice that my STM32 project is stored in a project directory; I have a .gitignore file to ignore those pesky object files from being committed, along with the Dockerfile and a top-level Makefile. You'll also notice that I have a .gitlab-ci.yml file. Later in the chapter, I will show you how to create this file; for now, just ignore it.

Name	Last commit
Project/makefile/Blinky	Updated Makefile
.gitignore	Add .gitignore
.gitlab-ci.yml	Update .gitlab-ci.yml
CHANGELOG	Add CHANGELOG
Dockerfile	Added STM32CubeMx generated files
Makefile	Updated Makefile
README.md	Initial commit

Figure C-3 A representation of the root directory after creating the STM32 project

COMPILING THE STM32 MAKEFILE PROJECT

We now have installed a Docker container with GCC-arm-none-eabi compiler and an example baseline STM32 project. Within the Docker image, which you should have opened in a terminal earlier, we want to navigate to the root directory of our STM32 project. For me, this is done using

```
cd Project/makefile/Blinky/
```

Once I'm in the directory, I can use the STM32 generated makefile to compile the project using

```
make all
```

The resultant output should look something like Figure C-4. You should notice that we successfully built our STM32 project from within a Docker image. While this doesn't seem super impressive now, this is precisely what we needed to do to start building our CI/CD pipeline.

```
=hard -specs=nano.specs -TSTM32L475VGTx_FLASH.ld  -lc -lm -lnosys  -Wl,-Map=buil
d/Blinky.map,--cref -Wl,--gc-sections -o build/Blinky.elf
arm-none-eabi-size build/Blinky.elf
   text    data     bss     dec     hex filename
  21232      32    9696   30960    78f0 build/Blinky.elf
arm-none-eabi-objcopy -O ihex build/Blinky.elf build/Blinky.hex
arm-none-eabi-objcopy -O binary -S build/Blinky.elf build/Blinky.bin
root@2ed2feef1d3d:/home/app/Project/makefile/Blinky#
```

Figure C-4 Successful compilation of the STM32 project within the Docker container

Note To exit the Docker container, type EXIT in the terminal.

CONFIGURING THE BUILD CI/CD JOB IN GITLAB

Now that we have a Docker environment successfully building our project, we are ready to create the build job in our Embedded DevOps pipeline. Obviously, before configuring the pipeline, you'll have to go through and create a GitLab account and follow their setup instructions. If you plan to follow along, you can find the instructions at https://docs.gitlab.com/runner/install/. Alternatively, you could use a tool like Jenkins and use the general process, but you'll be responsible for figuring out the details.

CREATING THE PIPELINE

Once you have your GitLab runner set up, log in to GitLab and navigate to your repo. On the left side of the interface, you'll find a CI/CD menu option that includes a pipeline menu. Click the pipeline. You should now see that you have options for creating your pipeline. GitLab, by default, does not have a C template option, but they do have a C++ option. While the C++ template is a good start, we want to understand the entire CI/CD setup process. Therefore, I recommend starting with the "Hello World with GitLab CI."

By clicking the template, GitLab will present a default .gitlab-ci.yml file for you to review and commit to your repository. Commit this file, and then let's explore its contents. The code can be seen in Listing C-6.

```
stages:          # List of stages for jobs and their order of
execution
```

```
  - build
  - test
  - deploy
build-job:       # This job runs the build stage first
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "Compile complete."
unit-test-job:   # This job runs in the test stage.
  stage: test    # Runs if build stage completes
successfully.
  script:
    - echo "Running unit tests... This will take about 60
seconds."
    - sleep 60
    - echo "Code coverage is 90%"
lint-test-job:   # This job also runs in the test stage.
  stage: test    # Run at the same time as unit-test-job
  script:
    - echo "Linting code... This will take about 10 seconds."
    - sleep 10
    - echo "No lint issues found."
deploy-job:      # This job runs in the deploy stage.
  stage: deploy  # It only runs when *both* jobs in the test
stage complete successfully.
  script:
    - echo "Deploying application..."
    - echo "Application successfully deployed."
```

Listing C-6 The “Hello World with GitLab CI” yml template file for a three-stage pipeline

The first thing to notice with our pipeline YAML file is that it creates three stages: build, test, and deploy. Each job created is assigned a stage in the pipeline to run. Each job also contains a script that specifies what commands should be executed during that stage. If you carefully examine the example YAML, you’ll notice the only calls made in the script are to echo and sleep. In time, we will replace these calls with our own.

> **Note** Each stage runs sequentially. If a job in the stage fails, the next stage will not run. If a stage has more than one job, those jobs will execute in parallel.

After committing the YAML file to your repository, you can navigate the CI/CD

pipeline menu again. You'll notice that it now displays the status of the CI/CD runs! For example, after the last commit, the pipeline began executing and, when completed, looked something like Figure C-5. Notice that we get an overall pass on the left, a repo description and hash, and icons for each stage in our pipeline. If one of the stages failed, that stage would be red and have an x in it to notify us where the pipeline failed.

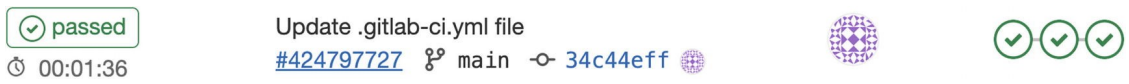

Figure C-5 The GitLab pipeline status shows that the pipeline passed all three stages successfully

You can click each check mark, or x mark, to review the output for each job. For example, if I click the first check mark that corresponds to the build process, you will see something like the details shown in Figure C-6. Notice that you can see our echo script commands being executed. We also get a confirmation at the end that the job succeeded. If the job had failed, this is where we would go to get the details about what went wrong.

```
19 Executing "step_script" stage of the job script                    00:01
20 Using docker image sha256:27d049ce98db4e55ddfaec6cd98c7c9cfd195bc7e994493776959db335
   22383b for ruby:2.5 with digest ruby@sha256:ecc3e4f5da13d881a415c9692bb52d2b85b090f38
   f4ad99ae94f932b3598444b ...
21 $ echo "Compiling the code..."
22 Compiling the code...
23 $ echo "Compile complete."
24 Compile complete.
26 Cleaning up project directory and file based variables               00:00
28 Job succeeded
```

Figure C-6 A partial output from the successful build job

CONNECTING OUR CODE TO THE BUILD JOB

Getting the example up and running is a significant first step, but we want to replace the "Hello World" scripts with connections to our code. To get our code connected, we're going to have a little bit of work to do.

The first step to connecting our code to the CI/CD pipeline is to make our Docker image available to GitLab. The easiest way to do this at this stage is to create a public image repository on Docker Hub. First, you'll need to go to https://hub.docker.com/ and create an account if you haven't already done so. Once your account is created, from your terminal, log in to Docker using the command:

```
docker login -u YOUR_USER_NAME
```

Enter your password, and the terminal should tell you that you are logged

in successfully.

Next, we want to tag and push our Docker image to the Docker Hub. You'll need to make sure that your image is tagged correctly. For example, if my Docker Hub username is beningo, then I want my image tag to be beningo/gcc-arm. If my username is smith, I want the tag to be smith/gcc-arm. You can tag your image using the following command in the terminal:

```
docker tag IMAGE_NAME YOUR-USER-NAME/IMAGE_NAME
```

From here, we just need to push the image to Docker Hub using the following commands:

```
docker push YOUR_USER_NAME/gcc-arm
```

You'll find that it will take a couple of minutes for the image to be pushed up to Docker Hub.

> **Beware** We are posting our Docker image to the public, which anyone can use! You may want to create a private repository and/or use your server for actual development.

The next step to connecting our code to the CI/CD pipeline is to update our YAML file. Two changes need to be made. First, we must tell GitLab what Docker image to use for the pipeline. We must add the code found in Listing C-7 to the top of our YAML file before the build job dictionary. Second, we need to update the build stage script to build our code using make! We can use make to build our code in several different ways.

```
default:
  image: beningojw/gcc-arm:latest
```

Listing C-7 The YAML file entry to tell GitLab to use our gcc-arm Docker image

First, we can update the YAML build stage script to match that shown in Listing C-8. In this case, we are using commands to cd into our project directory where the Makefile for the STM32 project is, cleaning it, and then running make all. Running make this way is acceptable, although it is not the most elegant solution. A better solution would be to create a high-level makefile in the root of your project directory. You can then use .phony to create make targets for

- Cleaning
- Building
- Testing
- Making, running, and pushing the Docker image
- Etc.

```
build-job:       # This job runs the build stage first
```

```
  stage: build
  script:
    - echo "Compiling the code..."
    - cd Project/makefile/Blinky/
    - make clean
    - make all
    - echo "Compile complete."
```

Listing C-8 Adding commands to move into the Blinky directory and execute make

Finally, we are ready to run our pipeline. The pipeline is executed when we commit code to the Git repo. Thankfully, we have just made changes to our YAML file! So, commit your code, log in to your GitLab project, and navigate the pipeline. You may need to wait a few minutes for the pipeline to run, but once the build process completes, you can click it and should see something like Figure C-7.

```
68    text           data     bss     dec     hex filename
69   21232             32    9696   30960    78f0 build/Blinky.elf
70  arm-none-eabi-objcopy -O ihex build/Blinky.elf build/Blinky.hex
71  arm-none-eabi-objcopy -O binary -S build/Blinky.elf build/Blinky.bin
72  make[1]: Leaving directory '/builds/beningojw/stm32-example/Project/makefile/Blinky'
73  $ echo "Compile complete."
74  Compile complete.
75  Cleaning up project directory and file based variables                    00:00
76  Job succeeded
```

Figure C-7 A successful build pipeline using our Docker image and custom code base

A great advantage to CI/CD is that you can be notified when a pipeline passes and fails. For example, Figure C-8 shows the contents of the email I received when I ran the build process successfully. If a build were to fail, I would also receive an email, but instead of a nice green happy email, I'd receive an angry red email! I don't like to receive failed emails because it tells me I messed up somewhere, and now I must go back and fix what I broke. The good news is that I know within a few minutes rather than days or weeks.

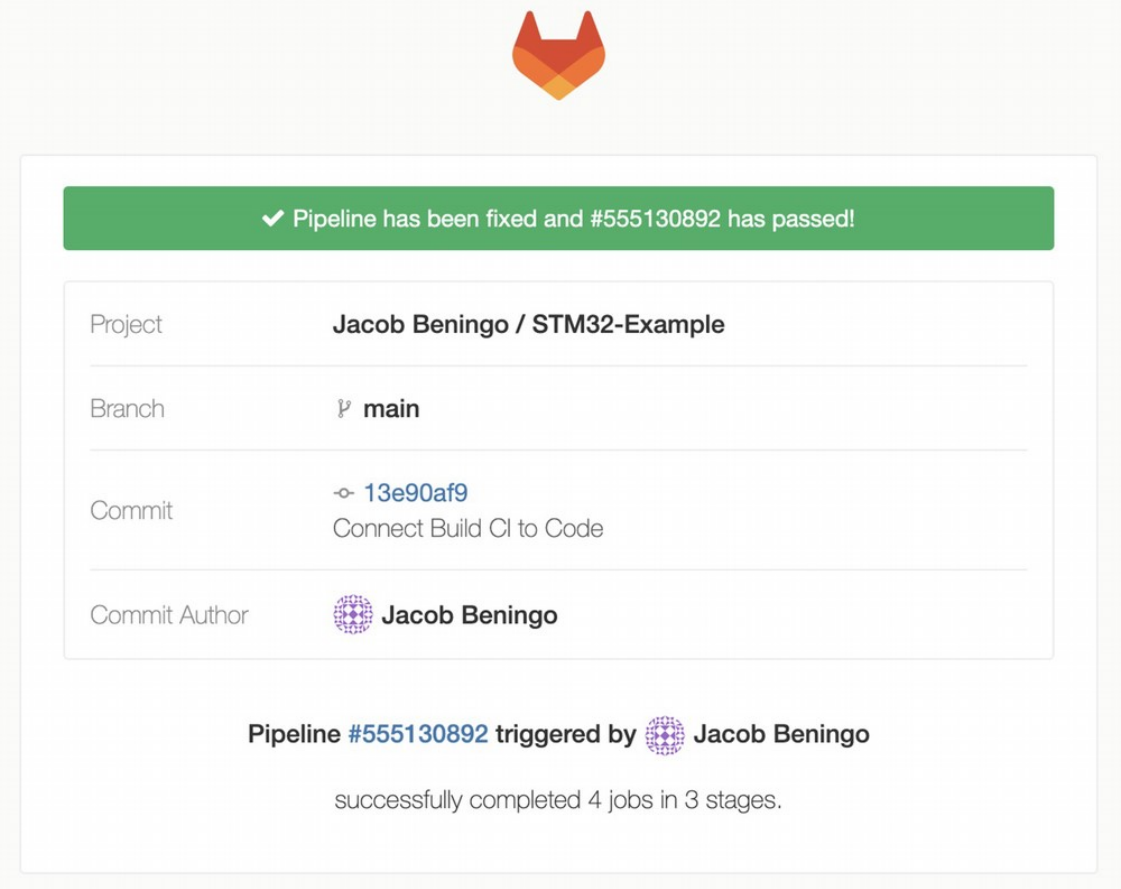

Figure C-8 The email notification received when the CI/CD pipeline runs successfully

BUILD SYSTEM FINAL THOUGHTS

Embedded DevOps has become a critical process for embedded software developers. Unfortunately, teams often put off using these modern processes because they fear them. They present an unknown to the team that requires some up-front research and development time to get it right. There's no reason to fear Embedded DevOps. So far, we quickly got a Docker container up and running to manage our build process and connected it to a CI/CD framework. Let's now look at how we can add CppUTest into the mix so that we can set up testing and regression testing in our CI/CD pipeline.

LEVERAGING THE DOCKER CONTAINER

The docker-compose run command causes Docker to load the CppUTest container and then make all. Once the command has been executed, it will leave the Docker container. In the previous figure, that is the reason why we get the ERROR: 2. It's returning the error code for the exit status of the Docker container.

> It isn't necessary to constantly use the "docker-compose run CppUTest make

all" command. A developer can also enter Docker container and stay there by using the following command:

```
docker-compose run --rm --entrypoint /bin/bash cpputest
```

By doing this, a developer can then simply use the command "make" or "make all." The advantage of this is that it streamlines the process a bit and removes the ERROR message returned when exiting the Docker image from the original command. So, for example, if I run the Docker command and make, the output from the test harness now looks like what is shown in Figure C-9.

```
root@e0384cff4bf3:/home/src# make
compiling MyFirstTest.cpp
Linking rename_me_tests
Running rename_me_tests
..
tests/MyFirstTest.cpp:23: error: Failure in TEST(MyCode, test1)
        Your test is running! Now delete this line and watch your test pass.

..
Errors (1 failures, 4 tests, 4 ran, 10 checks, 0 ignored, 0 filtered out, 0 ms)

make: *** [/home/cpputest/build/MakefileWorker.mk:458: all] Error 1
```

Figure C-9 The output from mounting the Docker image and running the test harness

To exit the Docker container, all I need to do is type exit. I prefer to stay in the Docker container, though, to streamline the process.

TEST-DRIVING CPPUTEST

Now that we have set up the CppUTest starter project, it's easy to go in and start using the test harness. We should remove the initial failing test case before we add any tests of our own. This test case is in /tests/MyFirstTest.cpp. The file can be opened using your favorite text editor. You'll notice from the previous figure that the test failure occurs at line 23. The line contains the following:

```
FAIL("Your test is running! Now delete this line and watch
your test pass.");
```

FAIL is an assertion that is built into CppUTest. So, the first thing to try is commenting out the line and then running the "make" or "make all" command. If you do that, you will see that the test harness now successfully runs without any failed test cases, as shown in Figure C-10.

```
root@001496277db5:/home/src# make
compiling MyFirstTest.cpp
Linking rename_me_tests
Running rename_me_tests
....
OK (4 tests, 4 ran, 9 checks, 0 ignored, 0 filtered out, 0 ms)
```

Figure C-10 A successfully installed CppUTest harness that runs with no failing test cases

Now you can start building out your unit test cases using the assertions found in the **CppUTest manual**. The developer may decide to remove MyFirstTest.cpp and add their testing modules or start implementing their test cases. It's entirely up to what your end purpose is.

INTEGRATING CPPUTEST INTO A CI/CD PIPELINE

Now that you've had the chance to play around a bit a test harness and experiment with Test-Driven Development a bit, it's a good time to discuss how to integrate CppUTest and your test cases into your CI/CD pipeline. Recall that when we set up our GitLab CI/CD pipeline, within our YAML file we had a section for a unit-test-job. We're now going to connect that job with our unit tests, but before we do that, we need to first add CppUTest to our Docker image.

ADDING CPPUTEST TO THE DOCKER IMAGE

In order to run our unit tests from within GitLab, we need our Docker image to have CppUTest installed. The easiest way to install CppUTest is to open our Dockerfile and add the code snippet that can be found in Listing **C-9**. Take a moment to examine the few lines of code. The first thing you should notice is that we are cloning the CppUTest git repository from the latest tagged version. We are then running three commands to configure and install CppUTest per the CppUTest documentation.

```
# Install and configure CppUTest
WORKDIR /home/cpputest
RUN git clone --depth 1 --branch v4.0 \
                    https://github.com/cpputest/cpputest.git
.
RUN autoreconf . -i
RUN ./configure
RUN make install
ENV CPPUTEST_HOME=/home/cpputest
```

Listing C-9 The Docker commands necessary to install CppUTest v4.0 from source

> **Tip**Before you rebuild your Docker image, navigate to `https://github.com/cpputest/cpputest` and look under tags. Replace v4.0 with the latest version of CppUTest.

If you were to run make image to rebuild the Docker image, you would discover that image fails to build. The problem is that we don't have the autoconf tool installed. You would then also find that you don't have libtool installed either. Within the Dockerfile, under the section where we are downloading and installing

Linux support tools, add the command to install autoconf and libtool. When you are done, your Linux support tool section should look something like Listing C-10.

```
# Download Linux support tools
RUN apt-get update && \
    apt-get clean &&  \
    apt-get install -y autoconf && \
    apt-get install -y libtool  && \
    apt-get install -y \
        build-essential \
        wget \
        curl \
        git
```

Listing C-10 The Dockerfile Linux support tool installation section

You can now rebuild the Docker image by running make image.

> **Beware** Whenever you update your Docker image, you'll need to make sure that you push it up to Docker Hub so that GitLab can access the latest image.

CREATING A MAKEFILE FOR CPPUTEST

Within our Docker environment, we are using make to manage our builds. Optimally, we would like to create a rule inside our makefile that we can use to call a secondary makefile that will manage all our tests. Before we create this rule, we need to create our CppUTest.mk file.

The best place to create your custom makefile for your project is to start with James Grenning's **cpputest-starter-project**[15] on GitHub. If you examine the repo, you'll find that there is a makefile that we can use as a template. I would recommend copying this file to your repo root directory and renaming it cpputest.mk. Once done, open the file and take a quick look through the contents of the makefile. As you can see, this can work as is, but for a custom application we will need to make several changes.

Let's start with specifying our output. Change the value of COMPONENT_NAME from rename_me to Tests/main. On the input side, you'll notice that we have to specify PROJECT_HOME_DIR and CPPUTEST_HOME. In my makefile, I've updated these lines of code to

```
CPPUTEST_HOME := /home/cpputest
PROJECT_HOME_DIR = /home/app
```

These are the locations where CPPUTEST and our project are located within the Docker container.

Next, we need to tell CppUTest where our source files are located. The source code can be found within the Firmware folder. We certainly don't want to have to list all the source files manually. Instead, we can leverage the shell find command to ease the burden. There are several different ways that we could provide CppUTest our source file. One way would be to define our sources in our top-level makefile. The problem with this method is that we may pull in files that we are not interested in testing such as middleware or vendor-supplied modules (although perhaps we should test these more than we typically do …). For now, we will be interested in only adding the source files in our application folder. The code to search our Firmware directory for headers, source, and assembly files can be found in Listing C-11.

> **Beware** Automatically searching for files is a great automation, but you can run into issues if you plan to use test doubles, mocks, and other test items.

```
SRC_FILES := $(shell find Firmware/Application -type f -name
'*.c')
```

Listing C-11 cpputest.mk updates to add our application source code to the test build

With our source files now added, we will want to specify our test directory. At this point, we don't have any files to add, but we do want the TEST_SRC_DIRS to include our Tests directory. Changing these couple lines of code to the following will do the trick:

```
TEST_SRC_DIRS += Tests
```

At this point, we are also not using any mocks, so these lines can be left as is. However, I would recommend enabling GCOV after the extension line. GCOV can be used to tell us how much test coverage we have on our source modules. GCOV can be enabled by including the following line:

```
CPPUTEST_USE_GCOV = Y
```

Next, we need to specify our include directories. By default, the cpputest.mk template has a bunch of additional include directories that we won't have in our own code. I would recommend deleting the lines of code and replacing them with the ones shown in Listing C-12. You'll notice that we are referencing a variable named HEADER_FOLDERS that we have not yet defined. In the top-level makefile, you can add the definition for the variable as follows:

```
SRC_DIR := Firmware
HEADERS := $(shell find $(SRC_DIR) -type f -name '*.h')
HEADER_FOLDERS := $(shell dirname $(HEADERS) | sort --unique)
INCLUDE_DIRS += $(HEADER_FOLDERS)
INCLUDE_DIRS += $(CPPUTEST_HOME)/include
INCLUDE_DIRS += $(CPPUTEST_HOME)/include/Platforms/Gcc
```

Listing C-12 The code necessary to include our include directories and those for CppUTest

A little bit further into cpputest.mk, you'll notice the following lines of code:

```
CPPUTEST_OBJS_DIR = test-obj
CPPUTEST_LIB_DIR = test-lib
```

These lines specify where the outputs from CppUTest will be placed. For now, I'm leaving them set as default, which means there will be a folder named test-lib and test-obj that will be created in the root folder of our repository when we run CppUTest. If you change the default paths, make sure you read the comments carefully. Some strange behavior can ensue.

The next section in the makefile allows us to customize what warnings and errors we would like to be enabled and disabled within the CppUTest build. I would recommend scheduling some time to examine this section more. For now, we will just leave the defaults as is.

Our makefile is now complete. The only thing missing is that we want to be able to run CppUTest from our top-level makefile. The code in Listing **C-13** shows how to create a rule to run CppUTest and also kick off gcov.

```
# Run CppUTest unit tests
tests: $(DIST_DIR)
	$(MAKE) -j CC=gcc -f cpputest.mk gcov
```

Listing C-13 The rule syntax to run CppUTest and gcov using our custom make-file

> **Tip**CC=gcc is used to tell CppUTest to use gcc. Without this, it's possible for confusion to occur between our C modules and the C++ test modules, resulting in a build failure.

Before we can run our tests, we need to add some application code to the Firmware/Application folder. If you would like, you can try out TDD yourself and write your own module, or you can look at Appendix D for a few example source modules. For this initial test, I will just be adding the led.h and led.c modules to Firmware/Application and then adding LedTests.cpp to the Tests folder. Once that is done, from a terminal, you can navigate to the root of the repo and execute the

following command:

```
make tests
```

If everything was configured correctly, which I'm sure it was, then you should see CppUTest and gcov run! Listing C-14 shows what you should expect to see as an output. Notice that our tests compiled, and we see the results of our tests and code coverage.

```
root@2267b7da4eee:/home/app# make tests
mkdir Distribution
make -j CC=gcc -f cpputest.mk gcov
make[1]: Entering directory '/home/app'
compiling AllTests.cpp
compiling LedTests.cpp
compiling MyFirstTest.cpp
compiling led.c
Building archive test-lib/libTests/main.a
a - test-obj/Firmware/Application/led/led.o
Linking Tests/main_tests
Running Tests/main_tests
....
OK (4 tests, 4 ran, 4 checks, 0 ignored, 0 filtered out, 0
ms)
/home/cpputest/scripts/filterGcov.sh gcov_output.txt
gcov_error.txt gcov_report.txt Tests/main_tests.txt
  Lines executed:100.00% of 6
100.00%   Firmware/Application/led/led.c
See gcov directory for details
make[1]: Leaving directory '/home/app'
root@2267b7da4eee:/home/app#
```

Listing C-14 The successful execution of CppUTest

TipIf your output shows that a test has failed, review the output, and resolve the failed test. Hint: Comment out the line of code!

CONFIGURING GITLAB TO RUN TESTS

Now that we have CppUTest up and running within our Docker environment, we can update our .gitlab-ci.yml to run our tests as part of the CI/CD pipeline. The adjustments that need to be made are straightforward.

First, open your YAML file and navigate to the area that defines the unit-test-job. You'll notice that we have a few echo statements that are simulating us running real unit tests. Remove those lines of code and replace them with the code found in Listing C-15. As you can see, all we do is add a call to make tests, and

everything goes from there!

```
unit-test-job:   # This job runs in the test stage.
  stage: test    # It only starts when the job in the build
stage completes successfully.
  script:
    - echo "Running unit tests ..."
    - make tests
```

Listing C-15 The changes to .gitlab-ci.yml that will run our tests as part of the CI/CD pipeline

With the adjustment made, it's now time to commit the code to the repo and let our pipeline run! After committing the code, go into the GitLab dashboard and open the pipelines. For me, it took about three and a half minutes for the full pipeline to run. Once completed, I could see that all the jobs passed successfully. Clicking the unit test job, you would see something like Figure **C-11** as the results. The output is essentially the same as when we execute it locally in our Docker image, except that now it is running on a Docker image on GitLab and as part of our CI/CD pipeline.

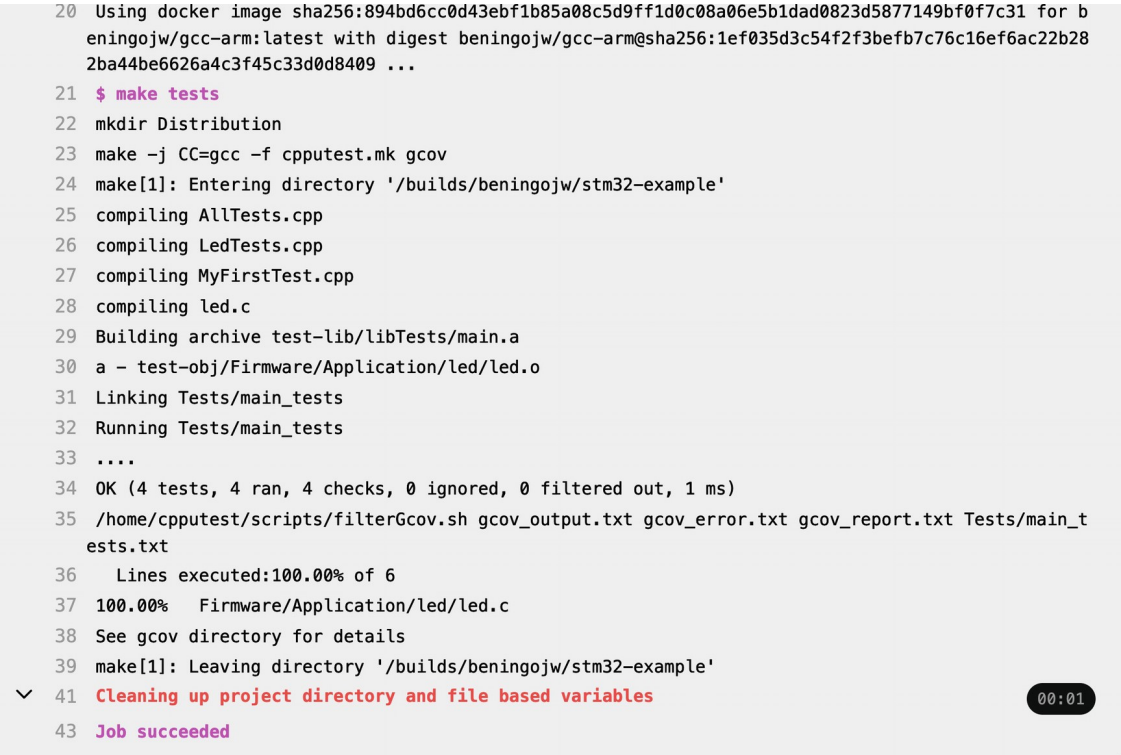

Figure C-11 The successful execution of the CppUTest unit tests on GitLab

UNIT TESTING FINAL THOUGHTS

Testing is a critical process to developing modern embedded software. As we have seen so far, there isn't a single test scheme that developers need to run. There are several different types of tests at various levels of the software stack that developers need to develop tests for. At the lowest lev-

els, unit tests are used to verify individual functions and modules. Unit tests are most effectively written when TDD is leveraged. TDD allows developers to write the test, verify it fails, then write the production code that passes the test. While TDD can appear to be tedious, it is actually a very effective and efficient way to develop embedded software.

Once we've developed our various levels of testing, we can integrate those tests to run automatically as part of a CI/CD pipeline. Connecting the tests to tools like GitLab is nearly trivial. Once integrated, developers have automated regression tests that easily run with each check-in, double-checking and verifying that new code added to the system doesn't break any existing tests. Let's now explore how we can deploy our software to the STM32 development board.

ADDING J-LINK TO DOCKER

When we deploy our compiled binary to our target device, we need to decide what mechanism we will use to program the target. There are several options available such as OpenOCD and SEGGER J-Link. As much as I love open source tools, I will use the SEGGER J-Link tool and executable.

To use J-Link, we must first install the J-Link tools as part of our Docker image. We can do this by adding the commands in Listing C-16 to the Dockerfile and then rebuilding the image using "make image." If you run the Docker container using "make environment," you can navigate to the /opt/SEGGER/JLink to see all the installed tools. Figure C-12 shows what you will find.

```
# Download and install JLink tools
RUN wget --post-data 'accept_license_agreement=accepted'
https://www.segger.com/downloads/jlink/JLink_Linux_x86_64.deb
\
&& DEBIAN_FRONTEND=noninteractive TZ=America/Los_Angeles apt
install -y ./JLink_Linux_x86_64.deb \
&& rm JLink_Linux_x86_64.deb
# Add Jlink to the path
ENV PATH $PATH:/opt/SEGGER/JLink
```

Listing C-16 Dockerfile commands to install and add J-Link to the path

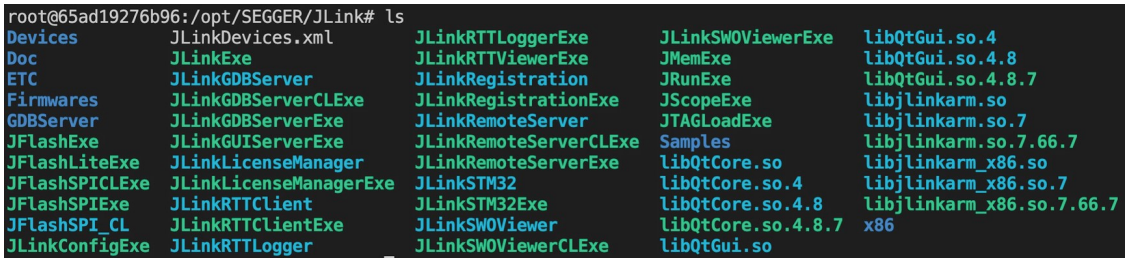

Figure C-12 The SEGGER J-Link tools installed in the Docker container

The tool that we will be using to flash the STM32 development board is JLinkExe. There are quite a few other exciting tools that come with the J-Link that are beyond the scope of our current discussion. I would highly recommend that you take some time to examine those tools and learn how you can use them to become a more effective embedded developer. For example, JLinkSWOViewerExe can be used to watch incoming SWO data during a debug session. JScopeExe can be used to trace an RTOS application.

ADDING A MAKEFILE RECIPE

Now that our Docker image has the necessary tools to program the development board, we need to add a recipe to the makefile that can use JLinkExe to program our binary image to the development board. A simple recipe that can be used is found in Listing C-17. Notice that we have named the recipe "deployToTarget." We have chosen this name over a simpler deploy to allow deployment to multiple environments. For example, we might want to deployToTarget, deployToProduct, deployToDevKit, or deployToSimulator. We aren't going to go into all these cases, but you can see how we want the flexibility to add more recipes as our pipeline becomes more sophisticated.

```
## Program the board with JLinkExe
deployToTarget: binaries
      JLinkExe -NoGui 1 -CommandFile
$(CONF_DIR)/program.jlink
```

Listing C-17 A makefile recipe for deploying a binary to a target device

We should also discuss a few additional points in the deployToTarget recipe. First, we make a call to binaries before we invoke a call to JLinkExe. The idea here is to make sure that we compile the code to the latest binary before we deploy it. We also don't have to make binaries before making deployToTarget manually. We just make deployToTarget, and everything is done for us.

> **Note** Contrary to popular belief, programmers aren't lazy! We have too much on our plates and must constantly automate and simplify our processes!

Next, the recipe executes the JLinkExe application in console mode without launching the graphical user interface. We are going to load the binary using a command file. A command file contains the commands for batch mode/auto execution.[16] The commands will perform a variety of functions, including

- Setting the target device
- Setting the J-Link speed
- Erasing the target
- Loading the binary
- Running the target

Listing C-18 shows an example command file for an STM32L475.

```
device STM32L475RE
si 1
speed 4000
erase
loadfile Firmware/Project/build/Blinky.hex
r
g
exit
```

Listing C-18 Example command file for an STM32L475

As part of the recipe, you'll also need to define CONF_DIR. To simplify things, I created a Config folder at the root of my repo. I placed the program.jlink file there. I then defined CONF_DIR := Config.

DEPLOYING THROUGH GITLAB

Now that we have a makefile that can successfully deploy our binary to our target device, we have just one more step until we can seamlessly deploy the device to our target; we need to update our GitLab pipelines. As you may recall, by looking at our .gitlab-ci.yml file, we have a deploy-job that isn't doing much more than pretending to be deployed to the target. Therefore, we can update our YAML file deploy-job to the code in Listing C-19.

```
deploy-job:      # This job runs in the deploy stage.
  stage: deploy  # It only runs when *both* jobs in the test
stage complete successfully.
  script:
    - echo "Deploying application..."
```

```
    - make deployToTarget
    - echo "Application successfully deployed."
```

Listing C-19 GitLab YAML file updates to deploy our binary to target as part of the CI/CD pipeline

At this point, you should be able to test that you can successfully program the target. Remember that the target must be connected to the computer on which you have your GitLab runner on. Otherwise, GitLab won't be able to detect the target and will lead you to a series of errors about not being able to connect to the J-Link or the target. The result will be a failed pipeline.

DEPLOYMENT FINAL THOUGHTS

We've just seen that deploying our firmware to a development board through CI/CD does not need to be overly complicated. The setup we just explored can be used to automate hardware-in-loop testing. Deploying software to a fleet of devices is more complicated; however, from the basics we have just explored, you should be able to build out an automated test suite to run on your development or prototype hardware.

APPENDIX D: HANDS-ON TDD

In Chapter 8, we discussed testing and Test-Driven Development (TDD). Appendix D is designed to show you the solutions that I came up with for designing a heater module using TDD. Your code and tests may turn out completely different than mine due to how you utilize the C programming language. In any event, Appendix D will show you an example solution.

THE HEATER MODULE REQUIREMENTS

The requirements provided for the heater module include

- The module shall have an interface to set the desired state of the heater: HEATER_ON or HEATER_OFF.
- The module shall have an interface to run the heater state machine that takes the current system time as a parameter.
- If the system is in the HEATER_ON state, it may not be on for more than HeaterOnTime before transitioning to the HEATER_OFF state. HeaterOnTime can be between 100 and 1000. Any number less than

100 results in 100; any number greater than 1000 results in 1000.

- If the heater is on, and requested on while active, then the HeaterOnTime shall reset and start to count over.

DESIGNING THE HEATER MODULE

So far, we have just a few simple requirements. From these requirements, we can begin to map out what our code is going to have to do. I often like to start by drawing out my design. Figure D-1 provides an example what the state machine design might look like. As you can see, the design doesn't have to be complicated. We have just a simple state diagram that shows the two major states and the conditions that cause transitions between the states.

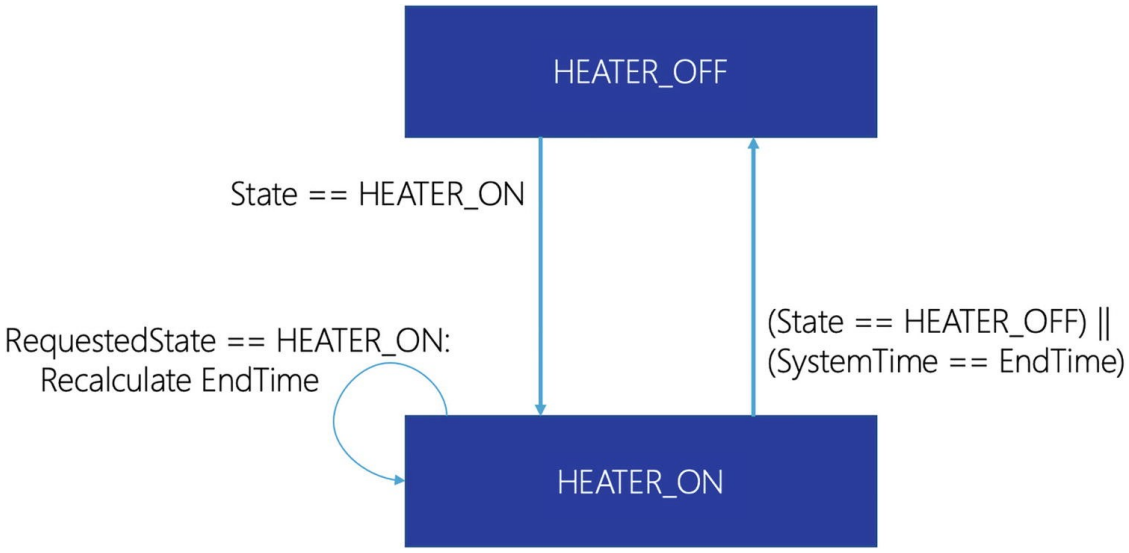

Figure D-1 The heater state machine design

Once I have my initial design in place, looking at the interfaces that I am going to need is a good next step. For example, after reviewing the requirements, I can see that there are three functions we are going to need:

- HeaterSm_Init
- HeaterSm_Run
- HeaterSm_Set

> **Note** I prefer to be verbose with my naming; however, I shorthand state machine as Sm. The interface could be HeaterStateMachine, but even with autocorrect, it feels too verbose for me. (There is also a 32-character limit in ANSI-C which I get worried about exceeding.)

You may be wondering why I have not defined a HeaterSm_Get function. The HeaterSm_Run function will return the current state. In fact, to avoid confusion, I

may map out an initial prototype of the interfaces as shown in Listing D-1.

```
void          HeaterSm_Init(uint32_t const TimeOnMax)
HeaterState_t HeaterSm_Run(uint32_t const SystemTime)
void.         HeaterSm_Set(HeaterState_t const State)
```

Listing D-1 Heater module interface

One of the tenants of TDD is that we don't write any code until a test forces us to. Technically, I have not written any code yet. I'm just mapping out what I think the interface for the module will be. The tests themselves will help me to flush these design elements out.

DEFINING OUR TESTS

Before I start to write my tests or my module, I also like to create a simple list of tests that will need to be performed on the module to make sure it meets requirements. For example, my initial test list for the heater module will look something like the following:

- The heater state initializes to HEATER_OFF.
- The heater can be set to the HEATER_ON state.
- The heater can be set to the HEATER_ON state and time out to the HEATER_OFF state.
- The heater can be set to the HEATER_ON state and commanded to the HEATER_OFF state.
- The current heater state can be retrieved from the run state.

You may wonder how I came up with this test list. The initial tests are listed based on reviewing the design. As I learned from James Grenning, my initial test list is not meant to be extensive. Typically, I will spend one to two minutes listing out all the tests that come to my mind immediately. I'm just trying to list out my starting point. As I develop my tests and code, additional tests may come to mind.

WRITING OUR FIRST TEST

For our first test, I will assume that you set up CppUTest in Chapter 8. You saw upon executing the test harness that you had a failed test caused by the FAIL macro in MyFirstTest.cpp. Copy the MyFirstTest.cpp and rename it to HeaterTests.cpp. Inside HeaterTests.cpp, rename the TestGroup to HeaterGroup. Also, set up the first test to now be in the group HeaterGroup and leave it as test1 for now. Make sure that the FAIL macro is included and run your test harness. You should see the following:

```
tests/HeaterTests.cpp:37: error: Failure in TEST(HeaterGroup,
test1)
        Your test is running! Now delete this line and watch
your test pass.
Errors (1 failures, 2 tests, 2 ran, 1 checks, 0 ignored, 0
filtered out, 0 ms)
```

We now know that our new test group has been added successfully. We can comment out our failed macro and get to writing our first test.

Let's start with the initial state of the heater. We want to initialize the system with the heater off. I'm going to start by renaming my test1 test case to InitialState. We get the current state of the system by running the HeaterSm_Run function. So, let's implement that now. Remember, for TDD we implement the minimum necessary for the test case. In this instance, I would create the enum for the HeaterState_t as shown in Listing **D-2**.

```
typedef enum
{
   HEATER_OFF,
   HEATER_ON,
   HEATER_END_STATE
}HeaterState_t;
```

Listing D-2 Definitions for the heater states

I would also implement the prototype and an empty function for HeaterSm_Run. I want the test case to fail first, so I will return HEATER_ON from the function as shown in Listing **D-3**. Now, I know that looking at Listing **D-3** is going to drive you crazy! It drove me crazy too when I first started with TDD. The minimum amount of production code required to make the test fail/pass is to just return the desired state! Many of you would want to create a variable right off the bat, but so far, that would be more code than is necessary for what the test cases are calling for.

```
HeaterState_t HeaterSm_Run(uint32_t const SystemTimeNow)
{
   return HEATER_ON;
}
```

Listing D-3 Minimum code required to pass the heater on test

Our very first test case will look something like Listing **D-4**. Notice we are just checking that the test returns HEATER_OFF. When we do run our test harness, initially the test should show us that it failed. The reason is that HeaterSm_Run is hard-coded to return HEATER_ON. Once you see that the test fails, update HeaterSm_Run to return HEATER_OFF.

```
TEST(HeaterGroup, InitialState)
{
```

```
  CHECK_EQUAL(HeaterSm_Run(0), HEATER_OFF);
}
```

Listing D-4 Test case to verify the initial state of the heater

Congratulations! You've just created your first test! We followed the TDD microcycle in this process. We added a test. We verified the test failed, which tells us that our test case would detect the failure case! If we just wrote a test that passes, we have no way of knowing if our test case will catch the failure case if something breaks. We made a small change to make the test pass. All the tests passed. At this point, there is no need to refactor. In fact, we are now ready to write our next text case!

TEST CASE – SETTING TO HEATER_ON

I know that leaving HeaterSm_Run to a hard-coded HEATER_OFF state is driving you crazy. Don't worry though. We will soon be forced to update this function. Let's tackle writing the test case to change the heater state from HEATER_OFF to HEATER_ON.

We start by adding a new test case to our test harness. I'm going to call mine StateOFF_to_ON. I'm going to start by implementing the test which can be seen in Listing D-5. I'll then run my tests. What you will discover is that there is a compilation error. It should tell you that HeaterSm_StateSet was not declared! The reason for this is that we have not yet added the interface. We can now go ahead and do so. (This is our small change.) We now should see the code compiling, but our test case is failing.

```
Test(HeaterGroup, State_OFF_to_On)
{
   HeaterSm_StateSet(HEATER_ON);
   CHECK_EQUAL(HeaterSm_Run(0), HEATER_ON);
}
```

Listing D-5 Example test for checking that the off to on transition works correctly

Our test case is failing because our HeaterSm_StateSet function is not setting the state of the state machine! We need a shared variable to hold the state between the HeaterSm_StateSet and HeaterSm_Run functions. We can do this by creating a static variable at module scope within heater_sm.c. The variable would be defined like this:

```
static HeaterState_t HeaterState = HEATER_OFF;
```

We can then set the state of the variable within HeaterSm_StateSet as shown in Listing D-6.

```
void HeaterSm_StateSet(HeaterState_t const State)
```

```
{
    HeaterState = State;
}
```

Listing D-6 Code to set the heater state

You can now run the test case. Unfortunately, you will find that it fails as shown in the following:

```
tests/HeaterTests.cpp:48: error: Failure in TEST(HeaterGroup,
State_OFF_to_ON)
        expected <0>
        but was  <1>
        difference starts at position 0 at:
<          1          >
                                                              ^
Errors (1 failures, 3 tests, 3 ran, 2 checks, 0 ignored, 0
filtered out, 0 ms)
```

The problem here is that our test case is failing because HeaterSm_Run has a hard-coded return value! We now need to make a small change to have HeaterSm_Run return the state of the variable HeaterState. Make that change now. Rerun your tests.

What on Earth is going on? My tests are still failing! The problem now is that our test harness gives no guarantee as to the order in which our tests are executing! You'll find that the test that fails now is InitialState! The reason is that State_OFF_to_ON runs and changes the state to HEATER_ON, and then the InitialState test runs and finds it is not HEATER_OFF. This is actually good! Our tests are discovering potential issues with our modules and our tests. What we need to do is after each test, put the heater module back into a known good state. In the TEST_GROUP section of our HeaterTests.cpp module, there is a function called teardown. If we update this function with the code listed in Listing **D-7**, the heater state will be restored after each test.

```
void teardown()
{
    HeaterSm_StateSet(HEATER_OFF);
}
```

Listing D-7 Code to reset the heater state at the end of each test

Now if you run your tests, you should see that all three pass as shown in the following:

```
OK (3 tests, 3 ran, 2 checks, 0 ignored, 0 filtered out, 0
ms)
```

Continue to work through your tests and develop your production code. Remember to take this slow and follow the TDD microcycle. I'm not going

to continue to walk you through all the tests. In the next two sections, you can find my final test cases and module code. After that, we will have a quick closing discussion.

HEATER MODULE PRODUCTION CODE

The header file for heater_sm.h will look something like Listing D-8.

```
/**************************************************************
* Title                 :   Heater State Machine
* Filename              :   heater_sm.h
* Author                :   Jacob Beningo
* Origin Date           :   07/30/2022
* Version               :   1.0.0
* Notes                 :   None
**************************************************************/
/** @file heater_sm.h
 *  @brief This module contains the heater state machine.
 *
 *  This is the header file for application control of the heaters.
 */
#ifndef HEATER_SM_H_
#define HEATER_SM_H_
/**************************************************************
* Includes
**************************************************************/
#include <stdbool.h>
/**************************************************************
* Preprocessor Constants
**************************************************************/
/**************************************************************
* Configuration Constants
**************************************************************/
/**************************************************************
* Macros
**************************************************************/
/**************************************************************
* Typedefs
**************************************************************/
/**
 * Defines the potential states for the heaters.
 */
typedef enum
{
```

```
    HEATER_OFF,          /**< Represents the heater off state */
    HEATER_ON,           /**< Represents the heater on state */
    HEATER_END_STATE     /**< Maximum state value used for bounds
checking */
}HeaterState_t;
/**************************************************************************
* Variables
**************************************************************************/
/**************************************************************************
* Function Prototypes
**************************************************************************/
#ifdef __cplusplus
extern "C"{
#endif
uint32_t HeaterSm_Init(uint32_t const TimeOnMaxValue);
HeaterState_t HeaterSm_Run(uint32_t const SystemTimeNow);
void HeaterSm_StateSet(HeaterState_t const State);
#ifdef __cplusplus
} // extern "C"
#endif
#endif /*HEATER_SM_H_*/
/*** End of File
**************************************************/
```

Listing D-8 The heater_sm.h file

The heater_sm.c module code will look something like Listing D-9.

```
/**************************************************************************
* Title                 :   Heater State Machine
* Filename              :   heater_sm.c
* Author                :   Jacob Beningo
* Notes                 :   None
**************************************************************************/
/**************************************************************************/
/** @file heater_sm.c
 *  @brief This module contains the heater state machine code.
 */
/**************************************************************************
* Includes
**************************************************************************/
#include <stdint.h>                 // For portable types
#include <stdbool.h>
#include "heater_sm.h"              // For this modules definitions.
/**************************************************************************
```

```
* Module Preprocessor Constants
*******************************************************************************/
/*******************************************************************************
* Module Preprocessor Macros
*******************************************************************************/
/*******************************************************************************
* Module Typedefs
*******************************************************************************/
/*******************************************************************************
* Function Prototypes
*******************************************************************************/
/*******************************************************************************
* Module Variable Definitions
*******************************************************************************/
/**
 * The current state value for the heaters.
 */
static HeaterState_t HeaterState = HEATER_OFF;
/**
 * Holds the state of the requested heater state
 */
static HeaterState_t HeaterStateRequested = HEATER_OFF;
/**
 * Tracks the time intervals since the HEATER_ON state was entered.
 */
static uint16_t TimeOnMax = 0;
/**
 * Stores the time to end heater control.
 */
static uint32_t EndTime = 0;
/**
 * Used to indicate if the heater off time needs to be updated.
 */
static bool UpdateTimer = false;
/*******************************************************************************
* Function Definitions
*******************************************************************************/
/*******************************************************************************
* Function : HeaterSm_Init()
*//**
* @section Description Description:
*
```

```
*  This function is used to initialize the heater state machine
parameters with
*  system level parameters.
*
* PRE-CONDITION:  None
*
* POST-CONDITION: The heater state machine parameters will be
initialized.
*
* @param              uint32_t TimeOnMax - The maximum time the heater
will remain on.
*
* @return             None.
*
* @see Heater_On
* @see Heater_Off
*
*****************************************************************/
uint32_t HeaterSm_Init(uint32_t const TimeOnMaxValue)
{
    if(TimeOnMaxValue < 100)
    {
        TimeOnMax = 100;
    }
    else if(TimeOnMaxValue >= 1000)
    {
        TimeOnMax = 1000;
    }
    else
    {
        TimeOnMax = TimeOnMaxValue;
    }
    return TimeOnMax;
}
/*****************************************************************
* Function : HeaterSm_Run()
*//**
* @section Description Description:
*
*  This function is used to progress and run the heater state machine.
It should
*  be called periodically.
```

```
*
* PRE-CONDITION:  HeaterSm_Init has been executed.
*
* POST-CONDITION: The heater state machine will be executed and either
the heater
*                 enabled or disabled.
*
* @param          uint32_t - The current system time.
*
* @return         None.
*
*****************************************************************/
HeaterState_t HeaterSm_Run(uint32_t const SystemTimeNow)
{
    static HeaterState_t HeaterStateTemp = HEATER_OFF;
    // Manage state transition behavior
    if ((HeaterState != HeaterStateRequested) || (UpdateTimer == true))
    {
        HeaterState = HeaterStateRequested;
        if (HeaterState == HEATER_ON)
        {
            EndTime = SystemTimeNow + TimeOnMax;
            UpdateTimer = false;
        }
        else
        {
            EndTime = SystemTimeNow;
        }
    }
    // Turn off the heater if we've reached the turn-off time.
    if(SystemTimeNow >= EndTime)
    {
        HeaterState = HEATER_OFF;
        HeaterStateRequested = HEATER_OFF;
    }
    return HeaterState;
}
/*****************************************************************
* Function : HeaterSm_StateSet()
*//**
* @section Description Description:
*
```

```
*  This function is used to request a state change to the heater state
machine.
*
* PRE-CONDITION:  None.
*
* POST-CONDITION: The HeaterStateRequest variable will be updated with
the desired state.
*
* @param          HeaterState_t - The desired heater state.
*
* @return         None.
*
* @see Heater_StateSet
*
*****************************************************************/
void HeaterSm_StateSet(HeaterState_t const State)
{
    HeaterStateRequested = State;
    // If the heater is already enabled, we need to let the state
machine know
    // that the time has just been updated.
    if (State == HEATER_ON)
    {
        UpdateTimer = true;
    }
}
/*************** END OF FUNCTIONS ***********************************/
```

Listing D-9 The heater_sm.c module

HEATER MODULE TEST CASES

The test cases that were used to drive the creation of the production code for the heater_sm module can be found in Listing **D-10**. Note, your tests may look slightly different, but should in general contain at least this set of tests.

```
#include "CppUTest/TestHarness.h"
extern "C"
{
  #include "heater_sm.h"
  #include "task_heater.h"
}
// Test initial state of state machine
// Test all state transitions of state machine
// Make sure the heater can turn on
```

```
// Make sure heater can turn off
// Turn heater on and let it time out
// Turn heater on and reset the count down
TEST_GROUP(HeaterGroup)
{
    void setup()
    {
    }
    void teardown()
    {
      HeaterSm_StateSet(HEATER_OFF);
      HeaterSm_Run(0);
    }
};
TEST(HeaterGroup, InitialState)
{
  CHECK_EQUAL(HeaterSm_Run(0), HEATER_OFF);
}
TEST(HeaterGroup, TimeoutValue)
{
  CHECK_EQUAL(HeaterSm_Init(0), 100);
  CHECK_EQUAL(HeaterSm_Init(500), 500);
  CHECK_EQUAL(HeaterSm_Init(1001), 1000);
}
TEST(HeaterGroup, State_OFF_to_ON)
{
   HeaterSm_StateSet(HEATER_ON);
   CHECK_EQUAL(HeaterSm_Run(0), HEATER_ON);
}
TEST(HeaterGroup, State_ON_to_OFF)
{
   HeaterSm_StateSet(HEATER_ON);
   CHECK_EQUAL(HeaterSm_Run(10), HEATER_ON);
   HeaterSm_StateSet(HEATER_OFF);
   CHECK_EQUAL(HeaterSm_Run(20), HEATER_OFF);
}
TEST(HeaterGroup, Timeout)
{
   // Turn the heater on for 100 counts
   HeaterSm_Init(100);
   HeaterSm_StateSet(HEATER_ON);
   CHECK_EQUAL(HeaterSm_Run(10), HEATER_ON);
```

```
    // Check at the 99 count that the heater is still on
    CHECK_EQUAL(HeaterSm_Run(109), HEATER_ON);
    // Check at the 101 count that the heater is now off
    CHECK_EQUAL(HeaterSm_Run(110), HEATER_OFF);
}
TEST(HeaterGroup, TimeoutReset)
{
    // Turn the heater on for 100 counts
    HeaterSm_Init(100);
    HeaterSm_StateSet(HEATER_ON);
    CHECK_EQUAL(HeaterSm_Run(10), HEATER_ON);
    // Reset the counter by setting the state to HEATER_ON
    HeaterSm_StateSet(HEATER_ON);
    CHECK_EQUAL(HeaterSm_Run(50), HEATER_ON);
    // Check at the 99 count that the heater is still on
    CHECK_EQUAL(HeaterSm_Run(149), HEATER_ON);
    // Check at the 101 count that the heater is now off
    CHECK_EQUAL(HeaterSm_Run(150), HEATER_OFF);
}
```

Listing D-10 The heater_sm module test cases

DO WE HAVE ENOUGH TESTS?

At this point, you should have created a module that roughly has around seven tests with around fourteen different checks. When I run the test harness, I get the following output:

```
OK (7 tests, 7 ran, 14 checks, 0 ignored, 0 filtered out, 0
ms)
```

The module seems to be doing everything that I need it to do. I used my tests to drive the production code I wrote. I used the TDD microcycle at each point to make small changes. I verified that every test failed and then made incremental adjustments until the tests passed.

The question that arises now is “Do I have enough tests?”. I used TDD to drive my code, so I should have all the coverage I need. I really should prove this though. The first check I can perform is to make a Cyclomatic Complexity measurement. This will tell me the minimum number of tests I need. Running pmccabe on the heater_sm.c module results in the following output:

```
Modified McCabe Cyclomatic Complexity
|   Traditional McCabe Cyclomatic Complexity
|       |    # Statements in function
|       |        |   First line of function
```

```
|       |       |       |   # lines in function
|       |       |       |       |  filename(definition line
number):function
|       |       |       |       |          |
3       3       6       85      17     heater_sm.c(85):
HeaterSm_Init
5       5       11      121     29     heater_sm.c(121):
HeaterSm_Run
2       2       3       169     11     heater_sm.c(169):
HeaterSm_StateSet
```

I should have at least three tests for HeaterSm_Init, five tests for HeaterSm_Run, and two tests for HeaterSm_StateSet.

This is where things can get a bit tricky. First, I only have seven tests in my test harness. Seven is less than ten, so I might be tempted to think that I have not reached my minimum number of tests yet. However, within those seven tests, there were fourteen checks. The fourteen checks more closely correlate to what Cyclomatic Complexity refers to as a test. In fact, Cyclomatic Complexity really is just telling us the minimum tests to cover all the branches.

A useful tool to help us close the gap on this is to enable gcov and see if we have 100% branch coverage. To enable gcov, in your cpputest.mk file, find the line with CPPUTEST_USE_GCOV and set it to

```
CPPUTEST_USE_GCOV = Y
```

Once you've done this, you can run gcov with the cpputest.mk file by running gcov afterward. For example, I have a makefile that has a "make tests" command that looks like the following:

```
test: $(DIST_DIR) ## Run CPPUTest unit tests
      $(MAKE) -j CC=gcc -f cpputest.mk gcov
```

When I run my test harness with gcov, the output states the following:

```
100.00%   firmware/app/heaters/heater_sm.c
```

As you can see, my 14 checks are covering 100% of the branches in the heater_sm module. In fact, I have four additional tests that are checking for boundary conditions as well.

TDD FINAL THOUGHTS

If you've followed along with this brief tutorial, congratulations! You've just taken your first steps toward using Test-Driven Development to improve your embedded software! As you've seen, the process is not terribly

difficult. In fact, even if it added just a few extra minutes, we now have a bunch of tests that can be run with each commit to ensure that we have not broken any of our code.

What's even more interesting, you've also seen in this example how we can create application code without being coupled to the hardware! We wrote an application code state machine, but we didn't care what I/O would control the heater. We simply wrote hardware-independent code to determine the state and then returned it. If we wanted to run this on-target hardware, we just need some code that takes the output from the state machine and knows which I/O line to control.

I hope that in this quick tutorial, you've seen the power of what TDD can offer you. I hope it has shown you how you can decouple yourself from the hardware and rapidly develop application code in an environment that is much easier to work in. Compiling code on a host and debugging is far easier than having to deploy to a target and step through the code. Yes, it's a bit different, but with some practice, you'll be using TDD efficiently and effectively to improve the quality of your embedded software.

Index

A

B

D

E

F

G

H

M

Q

R

S

T

U, V

W

X

Y, Z

Footnotes

1 https://courses.cs.washington.edu/courses/csep590/06wi/finalprojects/bare.pdf

2 www.cynet.com/network-attacks/privilege-escalation

3 https://powerdmarc.com/what-is-an-impersonation-attack

4 www.paloaltonetworks.com/cyberpedia/what-is-malware

5 https://csrc.nist.gov/glossary/term/man_in_the_middle_attack

6 www.ietf.org/id/draft-tschofenig-rats-psa-token-09.html

7 https://en.wikipedia.org/wiki/PSA_Certified

8 www.intrinsic-id.com/sram-puf

9 https://csrc.nist.gov/glossary/term/tampering

10 https://agilemanifesto.org/principles.html

11 https://docs.docker.com/get-docker/

12 https://docs.docker.com/compose/install/

13 www.st.com/en/development-tools/stm32cubeide.html

14 https://docs.docker.com/engine/reference/commandline/run/

15 https://github.com/jwgrenning/cpputest-starter-project

16 https://wiki.segger.com/J-Link_Commander#Using_J-Link_Command_Files

---