Introduction

► https://www.youtube.com/watch?v=WCoywUW5dwo *(YouTube video)*
In this training we are going to setup a proper embedded firmware development infrastructure so that you can build, test and deploy high quality embedded applications quickly.

This training takes you through the steps of implementing a modern continuous delivery build infrastructure for embedded firmware development.

This training is about YOUR repository setup and YOUR build infrastructure - and how we can make it better.

Instructor introduction

- 15+ years of software development experience
- Specializing in realtime embedded systems.
- CEO and founder of Swedish Embedded Consulting Group helping you build high performance embedded applications within IIoT, communications and industrial automation space.
- LinkedIn: https://www.linkedin.com/in/martinschroder/

Problems of firmware development

- Often difficult to test
- Often designed without portability in mind
- Often needs to be ported to multiple platforms
- Often difficult to guarantee quality without testing on physical hardware
- Expensive to test
- No well defined developer CI workflow

Firmware projects are complex - probably the most complex kind of software one can develop. All of this complexity needs to be managed somehow and we need ways of ensuring that the complex system does not break in unexpected ways.

The human brain can not comprehend all the details of this complexity all at the same time. This means that when a developer makes a change - he himself has no way of anticipating the ripple effects of that change through the software. Some component may now become broken and in the old way of building firmware you have no way of knowing that.

The new way of building firmware takes advantage of how most modern software is built - by leveraging continuous integration and continuous delivery and scripting every single step of the build process. In this training, I'm happy to say, we go even further and even script the tasks that a developer needs to complete and in what order.

With this infrastructure in place, you can put 10 developers working remotely on the same repository and you will have a smoothly operating team capable of releasing some small part of the system every day. By releasing, we mean an actual release - an actual built and fully packaged application. It may not be (and probably will not be) completed - but no matter what state it is in - it is fully tested and ready to be deployed.

This is the important part: we want to have continuously deployable application. To make this possible we need an automated build process with lots of automatic checks that offload the burden of checking from developers onto a build server.

In this training I will walk you through every step of creating a continuous delivery pipeline for your firmware project.

What we are going to do

In this training we will do our best to solve the typical problems of firmware infrastructure and at least point you towards a better architecture. Whether you choose to implement all of the covered methods or only some of them - you have a chance here to improve your firmware infrastructure.

The training is divided into 6 parts:

- **Setting up repository** - here we start from scratch and I show you how to configure the repository correctly.
- **Setting up GitLab CI** - here we create a basic CI pipeline for our firmware repository that will allow us to run the whole build process automatically.
- **Directory Structure** - we create a comprehensive directory structure that can accommodate for everything we may add to our project later.
- **Documentation Generation** - here we add a documentation generator form Zephyr that will generate a good looking reference manual for our project.
- **Test Infrastructure** - here we build out an automated test infrastructure using renode and CMock.
- **Release Generation** - here we generate a release with all files included - docs, software bill of materials and firmware.

what will be done in this course?

Setting up repository

In this section we setup a basic repository from scratch. The advantage of doing this from scratch is that you get to learn how the whole repository is set up. While we are following a similar setup to Zephyr, we are not starting with an already setup repository. Instead we start from zero.

In this section the following topics are covered:

- Adding the first robot framework scripts that will serve as the "definition of done" for this step.
- Adding required files: CMake files, Kconfig files, CODEOWNERS, gitignore etc.
- Adding a pre-commit hook that will run checks on changed files before each commit.
- Adding a repository init script that will set up a local repository and system after a checkout.

Setting up GitLab CI

Once the repository has been set up, we now need to configure gitlab so that we can move over to working with merge requests. To do this, we do the following:

- Configure a CI pipeline through .gitlab-ci.yml
- Create a proper task template (for creating tasks in gitlab)
- Build docker images so that we can run the pipeline locally through gitlab-runner.
- Add a merge request template
- Configure the repository for trunk based development.
- Add a check to ensure that merge request title contains a clickable reference to the ticket being solved.
- Add zephyr compliance checking scripts so that you can verify content of a merge request before merging it.

Directory Structure

Once we have a gitlab pipeline working, we can now work through merge requests and in this step we add the rest of the directory structure.

This includes:

- Adding an example application - so that you have a template to work from.
- Adding a custom board - so that we can build the application for custom hardware.
- Adding cmake library directory - this will be the place where we later put cmake helper scripts.
- Adding an example device driver - so that we have a template for device driver directories.
- Adding the documentation folder - we will add docs to it later.
- Adding a global include directory - for public interfaces of libraries and drivers visible across the whole firmware project.
- Adding an example library - so that we have a way of adding libraries shared between multiple applications.
- Adding the initial samples - so we can later boot them in renode to run user stories.

Documentation Generation

Once the full directory structure is in place it's time to add documentation generation. In this module we adopt Zephyr documentation generator so that it works in an external repository - such as the repository where we have our firmware code.

This involves:

- Moving the doc generator and adopting scripts so they build docs for our project.
- Adding robot framework scripts for verifying chapter structure - so that we can ensure that docs follow roughly the same structure.
- Linking the docs together so that we can generate a single PDF or HTML output.
- Adding the doc build process to CI and making everything pass all CI checks that we have setup earlier.

Test Infrastructure

In this section we focus on testing. There are three main types of tests that we need to add - unit, integration and system tests. Zephyr only has support for integration tests and a limited support for unit tests - so we have to add this. System tests are designed to test the final firmware, run it in a simulator (we will use renode in this training) and run user scenarios against it which verifies the integrity of the whole system.

- Adding unity - so that we can ditch ZTest and use a standardized testing framework.
- Adding CMock - so that we can mock any function in our unit tests and have the ability to verify our code for logical integrity.
- Adding renode support - so that we can simulate our firmware through renode and write scenarios in robot framework syntax.
- Code coverage checking - so that we don't allow any source file to be merged unless it has been fully tested.

Release Generation

In this section we will take everything that we have added so far and build a final release.

This involves:

- Semantic versioning - and its application to embedded build process.
- Adding software BoM generation - so that we can have a list of files that were included in the binary.
- Adding code to generate a final release archive.

Pitfalls

- Make sure you don’t go past a section you do not fully understand. This will only reduce your understanding even further (you can use discord channel to ask questions).
- Make sure you apply every piece of information before deciding whether you want to use it or not.

Success metrics

- An automated continuous delivery workflow.
- Main trunk always in working shape.
- Issues in code are caught before they get merged into main trunk.

After you have completed this training you should have a clear understanding of what steps go into creating a continuous delivery pipeline for an embedded project.

If you have any questions after you enroll in the training, you can always ask them in the discord community.

Summary

So here’s what we are going to do in this order:

- Configure repository and build process
- Configure gitlab ci
- Implement a robust and integrity checked directory structure
- Comprehensive documentation generation
- Test automation and simulation of firmware
- Release generation

Let’s get started!

Getting help

If at any time during this training you have any questions or need assistance, then use the open discord to get your questions answered:

- Discord direct link: https://discord.gg/bCSkK9xq
- Email: martin.schroder@swedishembedded.com

Chapter 1

Setting up your SDK repository

▶ https://www.youtube.com/watch?v=yBbt9qzgTNI *(YouTube video)*

Prerequisites

- Have at least some firmware development experience
- Desire to learn
- Not scared of using multiple languages (shell, Python, C, CMake etc.)

To make the most of this training you should preferably already have some experience with "broken" (or non-existent) CI systems and have at least a vague idea of what you have been missing. There are a lot of details in this training that you can use to solve common every day CI problems. You should be comfortable working with multiple languages at once and not be scared to use the right language for each job.

Problems we face

- We have a firmware to deliver and too little time to do it
- We need infrastructure that helps us deliver this firmware faster
- We need not just automation but also a workflow

It is our job to bootstrap a git repository and to bring it into at least a basic working state.

Our plan for solution

To begin solving this problem we need to at least bootstrap the repository with basic build scripts and CI so that we can then start adding everything else.

The first step in setting up a proper firmware development workflow is to create and setup a new git repository for our project. This is not just about creating the git repository but also making sure that we have all the basic files included - such as our zephyr manifest from which our workspace is going to be created.

Things we will do in this chapter are:

- **Creating the git repository** - we create and configure the git repository.
- **Adding all necessary files** - we add basic files that are necessary for initializing zephyr.
- **Adding our first robot framework script** - that will ensure we haven't forgotten anything and give us a way of defining that our first task is done.
- **Adding init script** - this script will run the zephyr commands necessary to checkout zephyr, install SDK and prepare our repository for development.

Setting up the git repository correctly from the start is important for a clean git history and continuous integration (we cover continuous integration philosophy in great detail in the continuous integration training which you should have gone through by now).

If we can make sure that a proper infrastructure is in place from the start then we can largely avoid the costly action of having to go back and redo things later.

Since this chapter is about creating the git repository itself, we will not cover anything beyond the basic setup. That we will do in subsequent modules of this training.

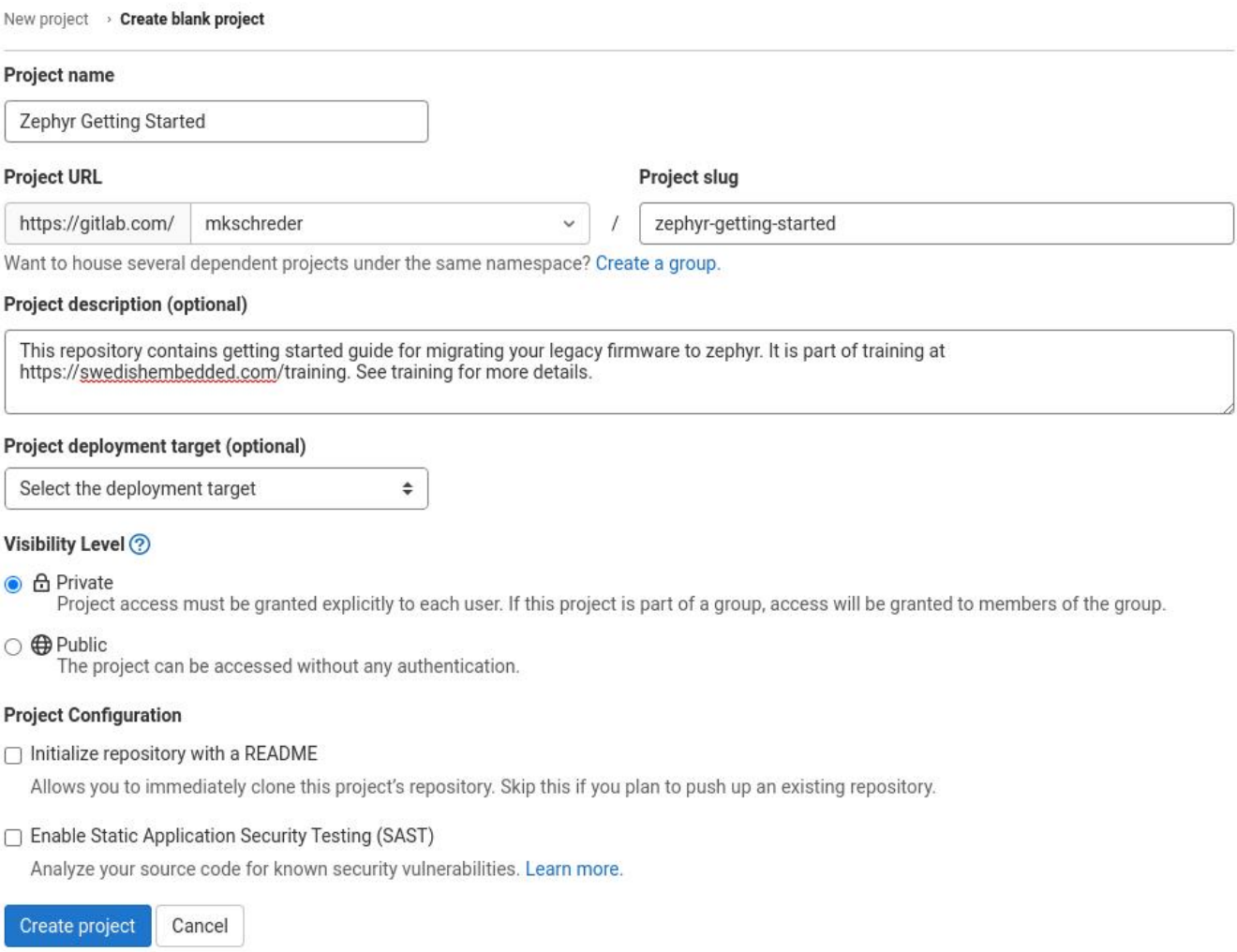
So let's jump right into it and create the repository where our new firmware project will come to life.

Setting up git repository

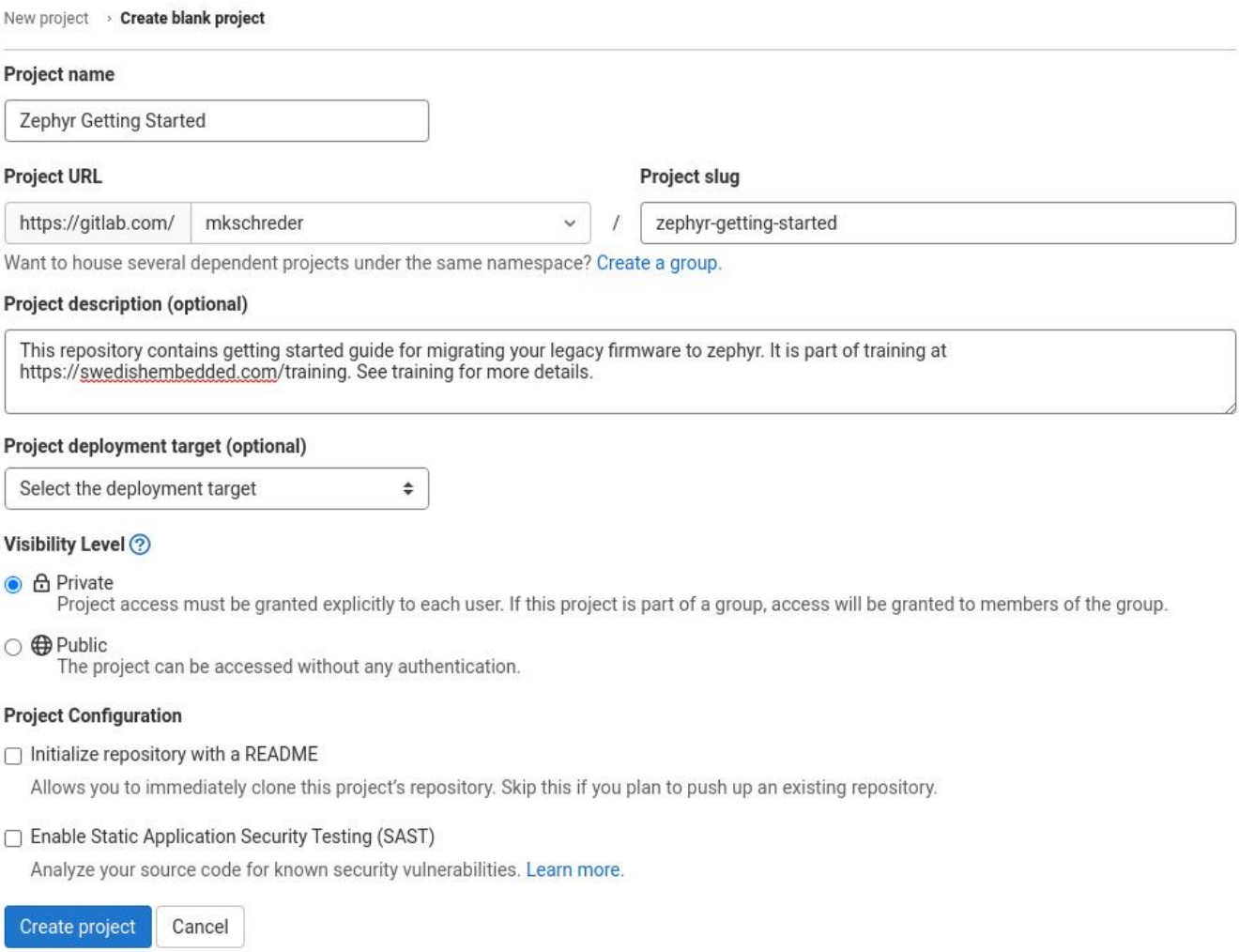
You can find source code for this step in the first commit of the course code repository: here

We will use GitLab as our main git repository management tool. Zephyr by default uses GitHub, and yes there are already scripts in Zephyr that implement large portions of the ci process in the GitHub way. However, that process is fairly complex so it helps greatly to look at how this process would be done on GitLab instead (we will still reuse Zephyr ci scripts - but our main ci process will be running on GitLab instead of GitHub).

Go ahead and create a new private repository in GitLab:



- Create new gitlab repository
- Configure its visibility



I usually prefer creating a blank repository and then populate it with README file and everything else.

Local workspace

Now create a directory on your machine for your workspace. You will checkout your git repository as a directory inside this workspace. If you are part of this training then you can request access to my git repository as well (but since I have to approve it manually, approval may take some time).

Create a local workspace:

```
$ mkdir zephyr-workspace && cd zephyr-workspace
```

Clone the repository into the workspace:

```
$ git clone \
    git@gitlab.com:user/zephyr-getting-started.git
$ cd zephyr-getting-started
```

In the future we will have this workspace setup:

```
zephyr-workspace
 ├─── bootloader ❶
 ├─── modules ❷
 ├─── tools ❸
 ├─── zephyr ❹
 ├─── zephyr-getting-started ❺
```

1. bootloader (by default mcuboot in Zephyr)
2. Zephyr modules
3. Zephyr tools
4. Zephyr kernel source code
5. Your source code

Creating a workspace directory is necessary because we will then checkout Zephyr files into the workspace directory alongside of our project. This is a very practical way of organizing our git repositories because we will have many different projects included into our firmware and having all of them as child directories of the project directory is not practical (too many files that make running "find" and "grep" commands cumbersome).

Let's change directory to our newly created repository and continue working on it:

```
$ cd zephyr-getting-started
```

For the rest of the training, we will work inside this directory most of the time. So if you see code that creates directories, it will always use paths relative to this repository directory.

Now it is time to actually start populating this repository with necessary files for your new project.

Directory structure

```
.
├── CMakeLists.txt ❶
├── CODEOWNERS ❷
├── Kconfig ❸
├── LICENSE ❹
├── README.rst ❺
├── scripts ❻
│   ├── checkpatch
│   │   └── typedefsfile
│   ├── checkpatch.pl
│   ├── init
│   └── spelling.txt
├── scrum ❼
│   ├── infrastructure
│   │   ├── readme.robot
│   │   └── repository.robot
│   ├── infrastructure.robot
│   └── variables.py
├── VERSION ❽
└── west.yml ❾
```

❶ this will be our main CMake file
❷ will be used for assigning code reviewers
❸ will be our main feature configuration file
❹ is important for project licensing
❺ will be the cover letter of our project
❻ will contain our CI and utility scripts
❼ will contain definition-of-done tests
❽ will contain global firmware SDK version
❾ will contain links dependency repositories

Robot Framework Tests

The first thing we want to do is define our "definition of done" for this module.

Throughout this training we will use robot framework for writing higher level tests and managing our TODO lists. This has the benefit of allowing us to ensure that what has been marked as "done" actually remains done and also to ensure that we never forget to do anything.

This is an extremely powerful approach of doing Scrum in practice. It allows us to not only define our epics and stories in a human readable form - but also to verify them as we go about completing them.

Let me show you how this works.

We will start with defining a directory called "scrum" where we will put our workflows:

```
$ mkdir scrum
```

Inside the scrum directory we will organize our stories and epics.

- **Epics** is defined as "a big chunk of work that has one common objective". In the case of this training, the epic may be expressed as "Properly set up infrastructure for a zephyr project" or simply "infrastructure".
- **Story** is a description of a software feature from a user's perspective. This defines how the work we do will deliver value back to our user.
- **Task** is a small piece of work that must be completed in full as means of implementing a user story or completing an epic. Tasks are usually small pieces of work with a granularity of a few hours.

Since there can be a huge number of tasks, it is often quite difficult to prioritize (and more importantly "reprioritise") the tasks.

We need a way to organize our epics, stories and tasks so that it is extremely clear what needs to be done next and we need a way to check that we have done everything without having to do this manually.

Robot framework makes this possible through the use of plain English definitions of tasks. Moreover, we don't need to define every single detail from the start - but rather we can go through the process incrementally - starting from the largest items and then breaking them down into smaller and smaller tasks.

In robot framework a task is defined by lowest level keywords that need to be defined as a way of marking a task as "done".

Syntax

- Independent tasks map to test cases
- Subtasks are expressed as keywords
- Test keywords can be written in python

Let's create an epic for our infrastructure that we are going to be implementing in this training:

```
*** Test Cases ***

Infrastructure has been setup
```

```
    Git repository has been setup
    GitLab CI has been setup
    Directory structure has been created
    Documentation generation has been setup
    Test infrastructure has been setup
    Release generation has been setup
```

Place this text into file under "scrum/infrastructure.robot".

We can now run this file using "robot" command:

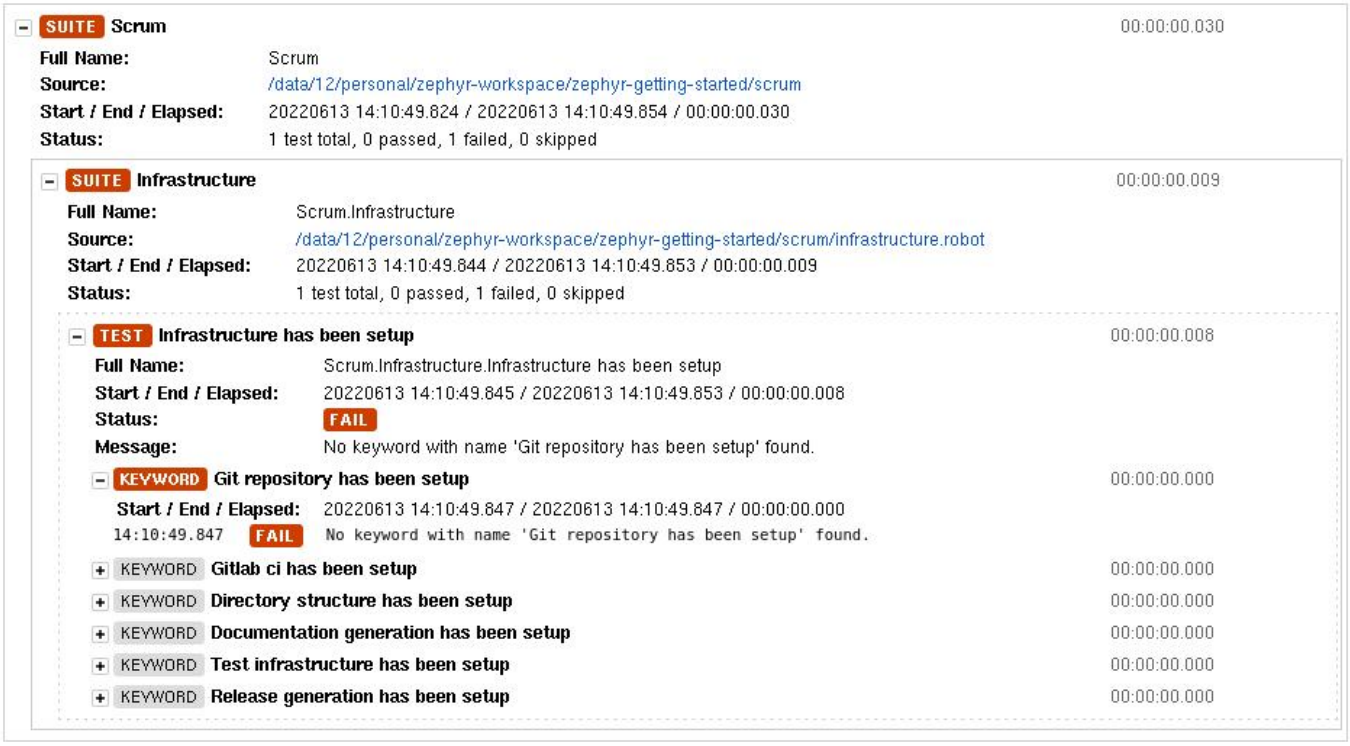
```
$ robot scrum/infrastructure.robot
```

We get an output that looks like this:

```
==============================================================================
Scrum
==============================================================================
Scrum.Infrastructure
==============================================================================
Infrastructure has been setup                                         | FAIL |
No keyword with name 'Git repository has been setup' found.
==============================================================================
Scrum.Infrastructure                                                  | FAIL |
1 test, 0 passed, 1 failed
==============================================================================
Scrum                                                                 | FAIL |
1 test, 0 passed, 1 failed
==============================================================================
```

If we look at the generated "log.html" file we will find a more detailed picture of what needs to be done:

Test Execution Log



We can now go ahead and define what we need to do in order to setup the git repository. For this we will define the "Git repository has been setup" keyword.

Let’s create a new file under "scrum/infrastructure/repository.robot" and define all the steps

iteratively progressing down to more and more detail:

```
*** Settings ***
Library  OperatingSystem
Variables  ${CURDIR}/../variables.py

*** Keywords ***

Git repository has been setup
    Zephyr repository has a west yaml file
    Zephyr repository has a top level CMake file
    Zephyr repository has a top level Kconfig file
    Zephyr repository has a license file
    Zephyr repository has a CODEOWNERS file
    Zephyr repository has a precommit hook
    Zephyr repository has a readme file
    Zephyr repository has a VERSION file
    Zephyr repository has an init script that sets up the workspace
    Zephyr repository has a gitignore file
    Zephyr repository has a robot framework script that checks repository structure

Zephyr repository has a readme file
    File Should Exist  ${PROJECT_ROOT}/README.rst

Zephyr repository has a top level CMake file
    File Should Exist  ${PROJECT_ROOT}/CMakeLists.txt

Zephyr repository has a top level Kconfig file
    File Should Exist  ${PROJECT_ROOT}/Kconfig

Zephyr repository has a license file
    File Should Exist  ${PROJECT_ROOT}/LICENSE

Zephyr repository has a gitignore file
    File Should Exist  ${PROJECT_ROOT}/.gitignore

Zephyr repository has a west yaml file
    File Should Exist  ${PROJECT_ROOT}/west.yml

Zephyr repository has a CODEOWNERS file
    File Should Exist  ${PROJECT_ROOT}/CODEOWNERS

Zephyr repository has a precommit hook
    File Should Exist  ${PROJECT_ROOT}/.githooks/pre-commit

Zephyr repository has a VERSION file
    File Should Exist  ${PROJECT_ROOT}/VERSION

Zephyr repository has an init script that sets up the workspace
    File Should Exist  ${PROJECT_ROOT}/scripts/init

Zephyr repository has a robot framework script that checks repository structure
    File Should Exist  ${PROJECT_ROOT}/scrum/infrastructure.robot
```

I have used a variables file here. Since robot framework is written in Python, you can use Python scripts to define variables. Here we can define a project root directory through a Python script like this:

```
#!/usr/bin/env python3
# File: scrum/variables.py
import os

# project root points to our project root folder
PROJECT_ROOT = os.path.realpath(os.path.dirname(os.path.realpath(__file__)) + "/../")
```

I have also used a very basic test to check if a task has been completed - we basically check whether the file exists. You can of course make your "definition of done" as complex as you want. You can even parse markdown or AsciiDoc documents and check content - or you can use libclang Python bindings to check internals of a C file as part of your robot scripts. The purpose is to make sure that we don’t forget to add anything that we would like to have there. Robot Framework scripts are perfect for this.

WEST Configuration

- A manifest containing external git repositories
- Supports hierarchical imports
- Extremely useful for a robust firmware build infrastructure
- Handles build orchestration
- Places external projects outside of current git repo (in contrast to git-submodules)
- Extendable using python

The first file we are going to add to our repository is the west manifest. This will be used by the zephyr west tool to initialize our repository.

We will later add more projects to this file, but for now we just want to tell west that we want to checkout zephyr (we can specify exact version) and we want to import this package - meaning that west will also look into zephyr west.yml and import all the projects that are in it.

Initialization

```
manifest:
  self:
    path: zephyr-getting-started

  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos

  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v3.0.0
      import: true
```

After adding the **`west.yml`** file, we can now initialize the workspace with all zephyr packages:

```
$ west init -l .
$ west update
```

This will result in a lot of packages being checked out to subdirectories of our parent directory (the workspace directory one level up). The workspace directory will now look like this:

```
$ tree -L 1 ..
zephyr-workspace
├── bootloader ❶
├── modules ❷
├── tools ❸
├── zephyr ❹
├── zephyr-getting-started ❺
```

❶ Bootloader(s)
❷ Vendor specific modules (stm32 hal etc)
❸ Zephyr tools
❹ Zephyr kernel source code
❺ Your source code

We can now proceed to add CMakeLists.txt and Kconfig file so that we can run basic build process.

CMakeLists.txt file

- Top level **`CMakeLists.txt`** includes infrastructure files
- Can define project wide compiler flags (such as warning levels)
- Unified build process across rtos, modules and our app
- Application level file defines how we will build our application

This file will ensure that we can include all of our custom drivers and libraries into our build. Anything we add into this file, will be globally visible to all of our applications and libraries. What I like to do here is to add extra GCC warning flags to the build.

We also add main project include directory here as well. We need to create this directory in our project as well (you may want to add that to the robot file too).

By default zephyr doesn't enable all warnings - but it should. Since the flags here will be visible to all code that is being built within our repository, we either need to patch zephyr or disable some of the warnings to make our code build. You can extend this list with more specific warnings if you want. Here is a full list of gcc warning options.

```
# Our project will have an include directory with public include files
zephyr_include_directories(include)

# Treat all warnings as errors
zephyr_compile_options("-Werror")
# Enable all standard warnings
zephyr_compile_options("-Wall")
# Extra warnings
zephyr_compile_options("-Wextra")
# Format options validation
zephyr_compile_options("-Wformat=2")
zephyr_compile_options("-Wformat-truncation")
# Detect some out of bounds array indices
zephyr_compile_options("-Warray-bounds")
# Detect uninitialized variables
zephyr_compile_options("-Wuninitialized")
# Detects passing null arguments into functions that do not check for it
zephyr_compile_options("-Wnonnull")

# Disable unused parameter warning
# Reason: zephyr pm functions fail to compile
zephyr_compile_options("-Wno-unused-parameter")

# Disable error on type limits (zephyr ASSERT)
# Reason: zephyr sys/assert does not compile
zephyr_compile_options("-Wno-type-limits")

# Disable error on comparison of integers of different signedness (zephyr)
# Reason: zephyr os/cbprintf does not compile
zephyr_compile_options("-Wno-sign-compare")

# Disable error on checking non-literal format (zephyr)
# Reason: zephyr drivers/serial/uart_native_posix.c does not compile
zephyr_compile_options("-Wno-format-nonliteral")

# Disable warning for any constants defined as 2.2 instead of 2.2f (as doubles)
```

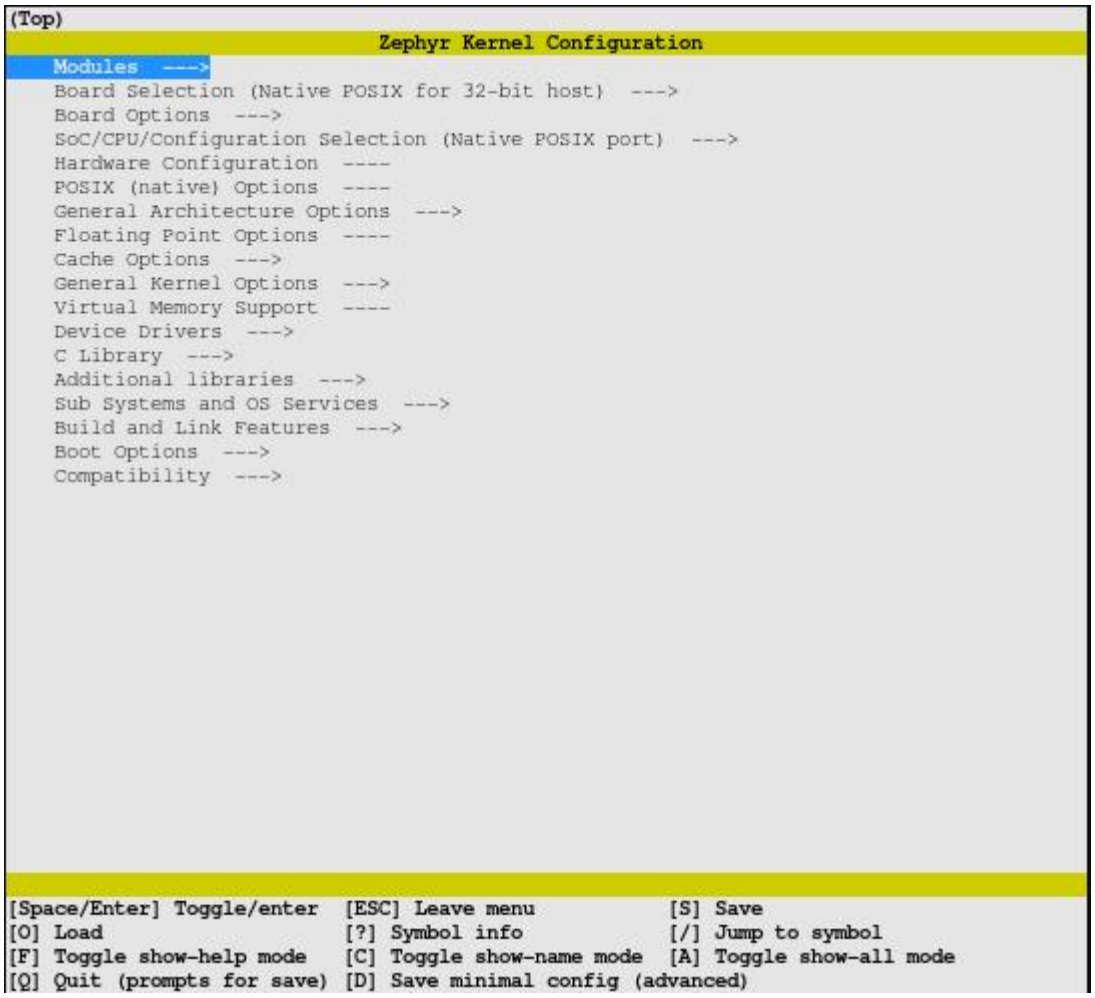
```
# Unfortunately has to be disabled due to zephyr not compiling with it
zephyr_compile_options("-Wno-double-promotion")
```

For now we don’t need to add any subdirectories. We will do that once we add drivers and custom libraries. Each application in our repository (including test applications) will have its own top level CMakeLists.txt file through which the above global file will be included automatically when we include zephyr package.

So we can leave this file as is for now and move on to our Kconfig.

Adding Kconfig file

- Defines "build configuration" parameters
- Allows complex parameter trees with dependencies
- Provides necessary flexibility for robust multiplatform development
- Independent language for software configuration
- Has its own set of utilities (kconfig)
- Popularized by the linux kernel project (kernel config)



Example

Example kconfig option definition:

```
config OPTION_NAME ❶
    bool "Description in menu" ❷
    depends on SOME_OTHER_CONFIG ❸
    default y if OPTA && OPTB > 1 ❹
    select SOME_OPTION ❺
    help ❻
        This multiline section contains
        help description shown when user
        presses "?" in menuconfig.
```

❶ Option name
❷ Option type and prompt
❸ Dependencies

4. Defaults
5. Automatic selections
6. Help text

Adding it to our repository

The Kconfig file will be used for including our project configuration files. We will be using Kconfig as our central configuration system. Kconfig configuration consists of Kconfig files spread out throughout our project.

- Drivers have Kconfig where we define driver options
- Application has a Kconfig with application options
- Libraries have their own Kconfig

We will use Kconfig for enabling and disabling every feature that we add to our project. In accordance with the rules of continuous delivery, we can commit and deliver a partially completed feature so long as it is not enabled in the final release application. We can use Kconfig to control when a feature is enabled and thus included into our release application.

Kconfig is also very powerful in terms of allowing us to specify dependencies between configuration options. We can specify what options need to be enabled first before (or available first) before the user can be allowed to enable another option. For example we can specify that SPI must be available before the user can enable a driver that uses SPI.

We also use Kconfig options inside our CMakeLists.txt files in order to include or exclude source code from our build.

At this point we will just add a simple empty Kconfig file that will act as a placeholder. Later we will add directives to this file to include subdirectories of our project repository. In this way we can have a single tree of configuration options where everything is included - both zephyr configs, modules, our own drivers and our own libraries.

The top level Kconfig file works in much the same way as the CMakeLists.txt file - it is not actually the "top level" Kconfig - rather each application has a Kconfig, that includes zephyr Kconfig - which then in turn includes our "top level" Kconfig which in turn includes all subdirectory Kconfigs inside our project (except applications).

We can still call it "top level" though since it is kind of a top level Kconfig for our libraries and drivers at least.

Let’s add an empty Kconfig file to our top level directory:

```
# This file will contain rsource directives for subdirectories
# For now it is empty.
# rsource "drivers/Kconfig"
```

We will later use "menuconfig" west directive to bring up a graphical interface where we can select all available config options. To get that working however, we will have to add at least a simple application. This will be done in subsequent modules. For now we just want to have all top level files in place.

Adding CODEOWNERS file

- Automatic reviewer assignment
- Division of responsibility
- Automatically parsed by GitLab and Github

The CODEOWNERS file consists of pairs of paths and gitlab users that should be assigned as reviewers of files under these paths. This gives you an automatic way of assigning code reviewers. So let's create a basic CODEOWNERS file.

We will base our initial file on the output of "tree" command. However, later we will add checks that verify that all files have been added to CODEOWNERS. Zephyr already has these checks - but we will move some of the zephyr checks later into our own project so that we have them as well.

Example codeowners

```
/CMakeLists.txt     @mkschreder
/Kconfig        @mkschreder
/LICENSE        @mkschreder
/output.xml     @mkschreder
/README.rst     @mkschreder
/scrum      @mkschreder
/west.yml       @mkschreder
```

Adding README.rst file

- Captures interest
- Provides pointers to further information and docs
- Provides quick start guidelines

The "README.rst" file will appear as the git repository "cover letter". It should act as a good introduction to your project (you can also view this file as a "sales letter" for your project and structure it as such - but structuring a sales letter is beyond the scope of this module for now).

The README page should accomplish the following:

- **Capture interest** - it is the first thing that a person will see when they land on your project.
- **Quick results** - it should help the user quickly attain a valuable result with your project (for example building a firmware they can actually flash and use by running just a handful of commands).
- **Link to official documentation** - it should show the user where to find official detailed technical documentation for your project.
- **Link to wiki** - it should link to any wiki related to your project where your team gathers information or where the user can join your community.

It should basically see all aspects of your project and provide pointers to the reader as to what to do next.

If we were to unpack this into a list of things that the README should have, then we can summarize it like this:

- **Niche specific result oriented headline** - a headline that grabs attention of the people who are most likely to respond.
- **Proof or screenshots** - if your project has a UI then this is where you put screenshots. You can also put proof of where your project is being used (if it is a public project).
- **Who is this for** - describe who is most likely to draw benefit from your project so that the reader of the README can feel "yes, that's me".
- **Core concept** - describe the core concept behind your project in a few paragraphs. Show diagrams if you have them. This is a bird's eye view of your "system" or "process".
- **Background story** - provide an explanation of what made you arrive at this solution. This should describe the "reason why" the process is the way it is. Provide a story of why you have arrived at this solution. This further builds on credibility.
- **Activities and unique insights** - this part describes activities involved in the process that your product helps to implement. All software is a way to automate some sort of process. Describe the details here.
- **Provide the next step** - you should always include information about the next steps that the user needs to take. This adds a natural flow to the information and the content becomes easier to read because you have provided links to where the reader needs to go next. If you don't provide some kind of next step, the reader becomes lost.

To verify that we have not forgotten anything, we can add a separate robot framework script for verifying the README. We can use simple grep commands to check for presence of

headlines which is a way for us to check that we satisfy a particular requirement. Of course this is very basic - but if we run the robot script and write each section in the order that they come then we will still remember to write all of them fully.

```
*** Keywords ***

Readme should follow sales letter structure
    Readme should have a niche specific headline
    Readme should contain proof and evidence
    Readme should make it clear who this is for
    Readme should describe the core concept
    Readme should describe the process and steps
    Readme should provide process summary
    Readme should contain transition into an offer
    Readme should specify who the offer is for
    Readme should outline benefits and outcomes
    Readme should list features and how the product works
    Readme should clearly state what reader should do next

Readme should have a niche specific headline
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Get Started With Zephyr

Readme should contain proof and evidence
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Screenshots

Readme should make it clear who this is for
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Who benefits from this the most

Readme should describe the core concept
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Zephyr Getting Started Guide

Readme should describe the process and steps
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Steps we cover

Readme should provide process summary
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Quick summary of the process

Readme should contain transition into an offer
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Do you really want to look at the code?

Readme should specify who the offer is for
    Rst Should Contain Headline
```

```
        ...  ${PROJECT_ROOT}/README.rst
        ...  Who would benefit from training

Readme should outline benefits and outcomes
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Benefits of doing the full training

Readme should list features and how the product works
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  What you will learn in the training that is not available here

Readme should clearly state what reader should do next
    Rst Should Contain Headline
        ...  ${PROJECT_ROOT}/README.rst
        ...  Attend the full training
```

As far as the "Rst Should Contain Headline" keyword, we can just implement it using the "grep" command. We do it like this in robot framework:

```
Rst Should Contain H1 Headline
    [Arguments]  ${RST}  ${HEADLINE}
    ${result}=  Run Process
    ...  grep  -Pzl
    ...  (?s)${HEADLINE}.*\\n\\*\\*\\*\\*\\*.*
    ...  ${RST}
    IF  ${result.rc} != 0
        Log To Console  ${result.stderr}
        Log To Console  Headline "${HEADLINE}" not present in "${RST}"
        Fail
    END
```

The above command uses grep in multiline mode to match title headline along with "*" on the next line. We need to add similar keywords for matching lower level headlines as well.

We can place the above into the "scrum/infrastructure/readme.robot" file along with our other keywords that process the README file.

Adding LICENSE file

- Establishes a distribution and collaboration pattern
- Establishes basic rules of engagement

You should always have a license file even if you are not working with an open source project. This makes it clear to everyone what rules apply to the repository.

In our particular case I will add the Apache-2.0 license file to my repository because I would like you to use this content (provided that you are a course participant) for your own needs and build upon it - so it makes sense for me to provide this code under the same license as Zephyr itself.

Gitignore file

- Ensures we do not accidentally push garbage or sensitive files
- Makes sure our git status is clean even if there are files present
- Ignore absolute paths: use forward slash
- Ignore pattern anywhere in tree: no forward slash

There are often folders (such as build folders) with files that you will never want to add to git. We can create a ".gitignore" file that will ignore these paths for us. Keep in mind that if you want to specify a path relative to your working directory, you should prefix it with a slash. If you just specify a file name then it will match that filename anywhere in your tree (which may not be what you want).

What to ignore

- Build artifacts
- Test output reports
- Complieance check logs
- GDB history
- Temporary files
- Python intermediate files
- West config

For zephyr project you can use the following gitignore file as a starting point. Add to it as your project evolves.

```
# Ignore build artifacts
/build*/
/twister-out*/
/html*/
/latex*/
/CMakeCache.txt
/doc/_build/

# Ignore test artifacts
/log.html
/report.html
/output.xml

# Ignore output of compliance checks
/Gitlint.txt
/Identity.txt
/Kconfig.txt
/KconfigBasic.txt
/Nits.txt
/checkpatch.txt
/compliance.xml
/pylint.txt

# Ignore gdb history
/.gdb_history

# Ignore editor swap files (vim, emacs etc)
```

```
*.swp
*~

# Ignore Python environment
.venv
__pycache__/

# Ignore generated west config
/.west/config
```

Pre-commit hook

- Run basic checks before commit
- Ensures we don't forget to run checks
- Drawback: prone to being annoying if takes longer than a few seconds!

Git provides us with hooks to enable running custom checks before allowing something to be committed. At the very least we want to start by running checkpatch on each commit that we put into git.

Checkpatch originally comes from Linux kernel, but zephyr uses it as well. Checkpatch checks for many little common issues inside the diff of the commit - such as wrong line endings, wrong formatting etc. It is a very useful first line of defense against bad code.

We can simply reuse the zephyr version of checkpatch by copying it into our project. Let's place it under the path as zephyr (you should try to replicate zephyr directory structure as much as possible in your project because it makes things much easier).

Checkpatch

- Copy it from zephyr along with supporting data

```
$ mkdir scripts
$ cp ../zephyr/checkpatch.pl scripts
$ cp ../zephyr/.checkpatch.conf .
$ cp ../zephyr/scripts/spelling.txt scripts
$ mkdir scripts/checkpatch/
$ touch scripts/checkpatch/typedefsfile
```

You also need to create the typedefs file and copy the checkpatch settings. The reason for this is that checkpatch comes from Linux and when we use it for zephyr project we really need to have a custom config that disables any Linux specific checks that we don't need. For now you can reuse the zephyr config.

We also create the spelling.txt file which checkpatch uses to check for common spelling errors (this is done on the whole patch - so it doesn't matter what file type the spelling error appears in). For now we just copy zephyr spelling file but later you can add to it.

Git hook

Now let's create our githook under ".githooks/pre-commit".

```
#!/bin/bash

# Check patch for errors and typos
git diff --cached | ./scripts/checkpatch.pl --no-tree || {
    echo ""
    echo "###################################################"
    echo "####    Checkpatch failed - commit aborted    #####"
    echo "###################################################"
    echo ""
    exit 1
}
```

One of the issues you will run into on the initial commit is that because you are committing the spelling file, your githook will not succeed since the spelling file contains typos. So for this

initial commit we need to ignore the githook. We can do this by committing with with "--no-verify" flag:

```
git commit -m "Add initial files" --no-verify
```

If you happen to need to make multiple changes to your initial commit then you can squash all commits on master into one using the soft reset approach (interactive reset will not work when you are dealing with initial commits):

```
git reset --soft id-of-first-revision-of-master
git commit --amend -m "Add initial files"
```

We also need to configure git to use our own githooks directory. We can do this by creating an init script and configuring hooksPath there. We will later use this init script for adding further initialization to our repository:

```
git config core.hooksPath .githooks
```

Adding VERSION file

- Holds project version
- Can be included in shell scripts
- Parsed by python scripts
- Used in CMake scripts and C code
- Single place for version

The version file in zephyr is used to generate compile time preprocessor definitions that define our project version. At this point in the project we have no way to verify that our versioning generation works so we will just add the version file with basic definitions - but we will wait to add actual cmake support for it until we add a basic application.

I prefer to use a version based on MAJOR, MINOR and PATCHLEVEL. This translates to version 0.0.1:

```
VERSION_MAJOR = 0
VERSION_MINOR = 0
VERSION_PATCH = 1
```

This will suffice for now.

Init script

- Initializes the build (not repository!)
- Updates important runtime dependencies based on module versions

Every project typically requires dependencies to be installed before the project can be built. I prefer to place an "init" script in the repository that installs all necessary dependencies. If we have an init script we can also run the same script in CI and install dependencies on docker in this way. So it's a multipurpose script that really help keep the process streamlined and predictable every single time.

We already have run a few commands that initialized our repository. Now we need to make sure that everything is scripted.

Here is the final init script that I have put together at this point:

```
#!/bin/bash

# abort on error (note that this is only a fallback. You should still check
# return codes every single time!)
set -e

# Configure git hook
git config core.hooksPath .githooks

# Install the west tool
pip3 install -U west

# Remove any existing west config (since we are reinitializing)
if [[ -d ../.west ]]; then
        rm -rf ../.west
fi

# Install apt packages (-qy means quet and say yes on all interactive questions)
sudo apt-get -qy update
# ... currently no packages here ...

# Initialize the repository
west init -l .
west update

# Install Python requirements
pip3 install -r ../zephyr/scripts/requirements-base.txt
pip3 install -r ../zephyr/scripts/requirements-run-test.txt
pip3 install -r ../zephyr/scripts/requirements-build-test.txt
pip3 install -r ../zephyr/scripts/requirements-doc.txt

# Install robot framework
pip3 install robotframework
```

Pitfalls of wrong repository setup

- Setting up repository properly early is very important
- Do not cut corners
- Any and all defects not detected from the start equals technical debt

If you don’t set up your repository correctly from the start then problems are missed and dealing with them later will require you to go back to old work and fix it. Thus the main pitfall to avoid is having to reopen work that should have been fully done.

This pitfall typically occurs when you setup your repository in a way that allows partially done work being committed without any checks. Work partially done is equivalent to "work not done at all" because one has to later go back and fix it. It creates a false perception of done - where a huge hidden cost of time and effort is present, but hidden from view.

Make sure you setup your repository in such a way that all work must be done before it gets shared with everyone else on the team (i.e. goes into main branch). Note that this is not to be misinterpreted as "no fully completed partially done features allowed - we allow partially done features so long as subtasks being added are being added in a fully completed state!)

Success metrics

By now you should have setup your repository and the scrum/infrastructure.robot script should be completing without errors. You should have implemented every step in that script fully. If you haven't done so already then go back and do it now.

Your repository should have a readme file, one commit on master with your initial files and properly setup githook that checks your full set of changes being done as part of one merge request for obvious errors.

Summary

In this module we have covered:

- Adding west.yml file
- Adding CMakeLists.txt file
- Adding Kconfig
- Adding README.rst file
- Adding LICENSE file
- Adding .gitignore file
- Adding CODEOWNERS file
- Adding pre-commit hook
- Adding VERSION file
- Adding init script

We have also committed our robot framework test that checks that all steps have been completed.

At this point we are ready to configure CI so that from this point onward we can start working through merge requests. For this we need to add a ci configuration file, configure at least two different pipelines and make sure that everything is working properly in gitlab.

In next module we will focus on precisely this. We will also make sure that our CI process prevents merge from happening if some of the scrum tasks do not pass. Currently we have to check this manually by running "robot scrum" command to execute all the robot framework scripts under "scrum" directory. In the next module we will add this check to the automatic CI process.

You can also see all changes in the git repository. You may need to request access to this repository.

Getting help

- Discord: https://swedishembedded.com/community
- Source code: https://gitlab.com/swedishembedded/training/zephyr-getting-started
- Email: martin.schroder@swedishembedded.com

Swedish Embedded Group offers consulting services that can greatly accelerate your firmware development process. If you don't want to do everything yourself then book a quick call with us and let's find out how we can help you get your firmware done faster.

Click here to visit calendar and pick your time for a call

```
Author: Martin Schröder, 14 Jun 2022
Email: martin.schroder@swedishembedded.com
Training: Zephyr Getting Started Training
Module: 1
```

Chapter 2

Setting up Gitlab CI

► https://www.youtube.com/watch?v=sDfed_OEvXE *(YouTube video)*

Progress so far

In the last module we did the first steps - which are the very basics of getting your repository properly setup.

We have:

- Setup basic build system
- Configured git hooks
- Added basic integrity checking
- Configured codeowners
- Added west configuration

Problems we face

Currently, your new repository doesn’t yet have a way to automatically check each merge request for compliance.

- No git workflow
- No automatic builds
- No automatic test runs
- No way to prevent broken trunk

What we are going to do

As part of the CI process we would like to enable the following:

- Multistage build pipeline: check, build, test
- Integration with custom docker images
- Enforcement of linear history and clean trunk
- A workflow for code review

This will give us:

- **Automatic compliance check before merge** - so that we check all the formatting of our code and formatting of the patch itself (i.e. the merge request branch as a whole).
- **Automatic build of all our applications** - so that we know that all our code builds before it gets merged into trunk.
- **Automatic test execution** - so that we know that all of our tests both build and run.

As the first step towards our solution we first need to configure gitlab CI.

Gitlab CI steps

Gitlab CI is configured through the **`.gitlab-ci.yml`** file placed inside our git repository. However, that's the easy part. We also need to write the scripts that will automatically build our code and check it.

Just like in the last module, let's start by mapping out what we need to do as a robot framework outline and then handle each task in order throughout this module.

The scope of this training can be defined as follows:

```
Gitlab ci has been setup
    Gitlab task template has been added
    Gitlab CI has a fully configured build pipeline
    Gitlab repository has merge method set to fast forward merge
    Gitlab repository has mandatory squashing
    Gitlab repository has merge checks enabled
    Gitlab repository has squash commit template
    Gitlab merge request template has been added
```

We are going to put this along side of all our other scrum files and continue extending this script **`scrum/infrastructure/gitlab-ci.robot`**.

Repository settings

First we need to configure some important settings in our repository.

Problems with defaults

- Non-linear history
- Commit message templates not setup
- CI not mandatory by default

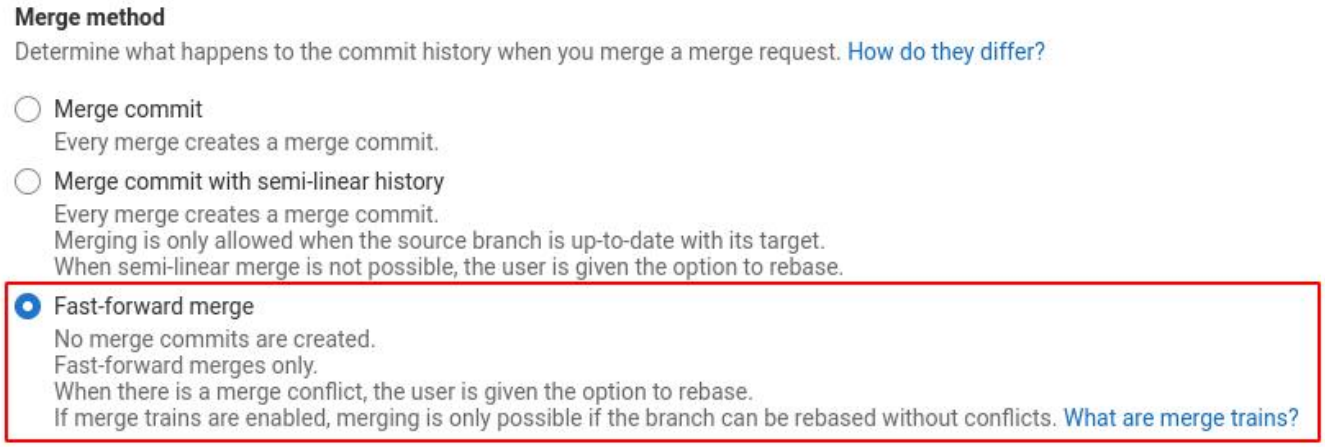
The default configuration is not complete. We need to set this up properly first.

Configuration

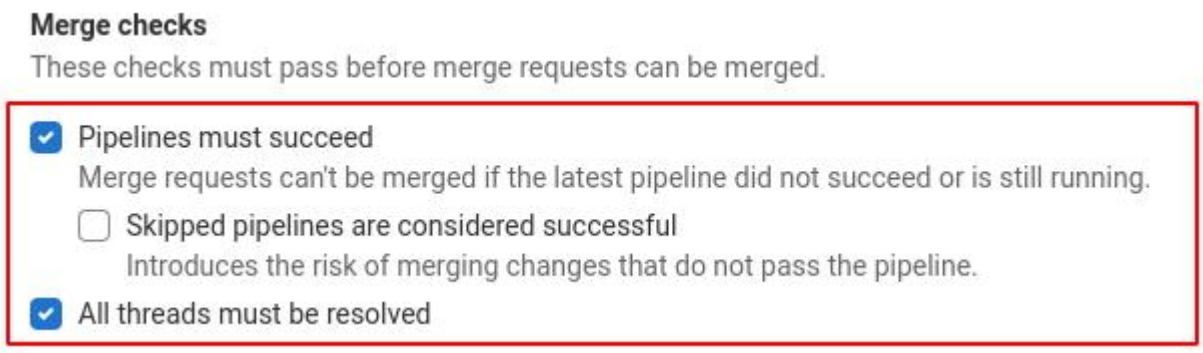
Under **`Settings > Merge requests`** do following:

- Set to fast forward merge
- Squash commits when merging: encourage
- Merge checks: enable
- Squash commit message: add description

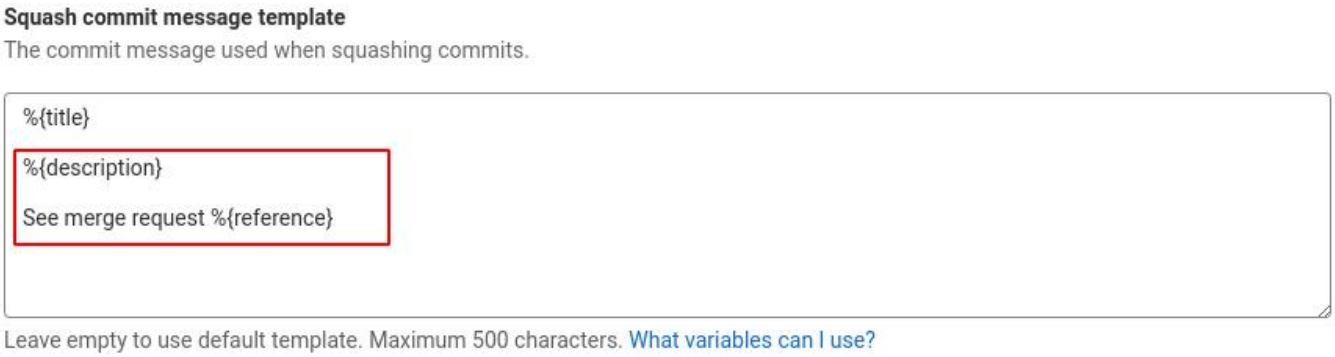
First we set our merge method to fast forward so that we don't get any merge commits. This will give us linear history. I mention the benefits of having linear git history in my trunk based development training - so make sure you check it out as well.



Next we want to enable merge checks (if you haven't done so already):

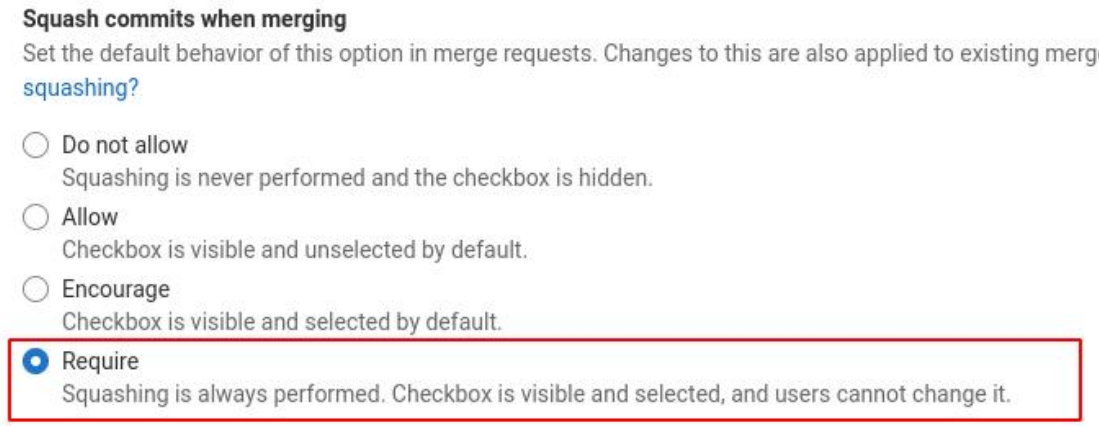


And the squash commit template:



Also make sure that you either enable mandatory squashing or set this option to "encouraged". This will ensure that developers don't need to worry about "fixup" commits on their work in progress branches. These will all be squashed into a single commit in trunk.

The only reason you may want to set this option to "encourage" instead of "require" is if you have multiple repositories where you would like to cherry pick commits between them. In such a scenario, mandatory squashing will result in fully perfect commits being squashed again - which is not what we want. When you do cherry picking you will therefore want to simply rebase and fast forward instead.



Adding a task template

The first thing we are going to do is add a task template:

Task template gives us several benefits:

- Makes it easier to remember information to include
- Helps when prioritizing and assigning tasks

The tesk template is added into the file **`.gitlab/issue_templates/task.md`** and contains default markdown text to be used when creating a new task.

The task template should at the very least contain following information:

Content of task template

- What needs to be done
- Why it is needed
- What is out of scope for this task
- Additional information (links, screenshots etc).

Source

```
## What needs to be done

Describe what needs to be done

## Why we need it

Describe in words or in RobotFramework syntax what items need to be completed
and verified.

## What is out of scope

Describe a list of things that may be tempting to do as well but which are
currently out of scope for this task.

## Additional information

Links to documentation, pages and other resources that help with
implementation.
```

This will later appear as a drop down when creating a new issue in the repository.

Merge request template

We also need to add a merge request template to our repository. This template will help remind developers what information they need to include into each merge request description. Since merge request descriptions are also used for main trunk git commit message content, these need to be in good shape.

The merge request template helps us achieve:

- Consistency of git commit messages
- Simplify code review process by providing information

The merge request template goes along side of task template into **`.gitlab/merge_request_templates/default.md`** file.

Content of MR template

- Summary of changes
- Proof of results
- Unfixed issues and limitations
- Ticket references

Source

```
## Summary

List what has been done.

## Results

Show what the final result looks like.

## Known issues/limitations

List known issues with current solution and limitations that have not been
handled.

Closes #xxx
Related to #xxx
```

CI Job Configuration

We are going to setup three automated CI jobs:

- Check: integrity checking
- Build: compilation
- Test: instrumented compilation

These build jobs will run in sequence for each and every merge request. If any of these jobs fail then the new changes will not be allowed to be merged.

To define these jobs, we are going to add three sections to our **`.gitlab-ci.yml`** file:

```
stages:
 - check
 - build
 - test

check:
    stage: check
    rules:
        - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    script:
        - ./scripts/init
        - ./scripts/check
build:
    stage: check
    dependencies:
        - check
    rules:
        - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    script:
        - ./scripts/init
        - ./scripts/build
test:
    stage: check
    dependencies:
        - check
    rules:
        - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    script:
        - ./scripts/init
        - ./scripts/test
```

Build from scratch

- Builds everything on blank ubuntu image
- Ensures install-sdk works as expected

We are also going to add a separate job that we will schedule to run nightly. This job will start with a clean Ubuntu image and install all of the necessary tools from scratch. This job is for testing the install-sdk script:

```
build_from_scratch:
    image: ubuntu:latest
    stage: build
```

```
    rules:
        - if: $CI_PIPELINE_SOURCE == "web"
        - if: $CI_PIPELINE_SOURCE == "schedule"
    before_script:
        - bash -x ./scripts/install-sdk
        - bash -x ./scripts/init
    script:
        - robot scrum
        - bash -x ./scripts/check
        - bash -x ./scripts/build
        - bash -x ./scripts/test
```

We make sure that this stage will run when CI is triggered either through the web interface of from a scheduled task (configurable in Gitlab interface).

Docker image

Zephyr provides an existing docker image which is used for all CI jobs that run as part of Zephyr infrastructure. However, for a custom project, using the Zephyr docker image is not enough.

Your project will likely have additional dependencies that will need to be installed before build process can complete and the proper way to manage these dependencies is with the use of a custom docker image.

- Use existing SDK image as base
- Build it locally on your build machine
- Use it for running your CI jobs
- Add additional packages to it as necessary

Swedish Embedded Platform SDK comes with two default docker images you can start with:

- CI Build docker image.
- Develop docker image.

Both are based on original Zephyr images but with many more tools added (such as clang-tidy, cmake-format, robotframework etc).

Building Docker image locally

- git clone https://github.com/swedishembedded/develop.git docker-image
- cd docker-image
- ./scripts/build

To create your own docker image, fork that repository and modify the Dockerfile adding your own packages to it.

Once you run **`./scripts/build`**, docker will build the image from your definition and store it locally on your machine. We can then register a gitlab runner locally and connect it to your repository so that any merge requests that are created will be executed on our local custom docker image.

Configuring CI

- Gitlab CI runner uses local docker image or pulls remote
- We can set "pull_policy: if-not-present" to always use local image if available

Once your docker image is built locally (or on your build machine), you should see the following:

```
$ docker image ls
swedishembedded/build    latest            b3443a32a546   6 days ago     11.8GB
```

The full name of the image in this case is **`swedishembedded/build:latest`**.

Runner installation

- Go to **`Settings > CI/CD > Runners`**
- Follow instructions to register local runner

- Once connect, your runner will now use docker image in your gitlab-ci.yml

To connect gitlab to our local build machine docker image we need to install gitlab runner locally and configure it with the gitlab key provided by gitlab.

Got to **`Settings > CI/CD > Runners`** in either your project or your Gitlab group and follow the instructions outlined there.

You can register a runner for the whole project group - which is very convenient if you want to use the same runner for multiple projects.

Another thing you need to do is disable public runners by default for your embedded projects. This will avoid your pipeline waiting for a public runner to pull the docker image. If you disable public runners and use only your custom runner then your pipelines will start instantaneiously.

Keep in mind that if your project is public, anyone with ability to create merge requests for your project will be able to run code on your runner (inside docker but still). So for public projects you would want to use public runners.

Once your runner is configured and connected to gitlab with the api token, it will be getting jobs from gitlab based on the project gitlab CI files. The local docker image on your build machine will be used to build your project.

First, install the latest version of gitlab runner:

```
sudo curl -L --output /usr/bin/gitlab-runner \
    "https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64"
```

Since your docker image should be named "swedishembedded/build:latest" you can use the local image when running the pipeline. To do this you must run the pipeline like this:

```
gitlab-runner exec docker \
    --docker-pull-policy="if-not-present" merge_request_pipeline
```

Once you have this working, you can register your local runner with gitlab CI. Go to project settings, then CI/CD then open the Runners tab. There you can get your runner registration token. Then run:

```
sudo gitlab-runner register \
    --url https://gitlab.com/ --registration-token $REGISTRATION_TOKEN
```

Choose "docker" as runner type.

Once registered, remember to add 'pull-policy = "if-not-present"' to /etc/gitlab-runner/config.toml under "runners.docker" section. This will make this setting enabled for all pipelines.

CI Config

- Set the new docker image for all CI jobs
- Use tags to pick runners

We can now configure our GitLab CI to use this image by adding it as default image to our **`.gitlab-ci.yml`** file:

```
image:
    name: swedishembedded/build:latest
    pull_policy: if-not-present
```

```
default:
  tags:
    - docker
```

You can also specify the image for each job individually, but doing it in global scope of the yml file makes all of your jobs use the same image. Note that only artifacts are preserved across jobs - that’s why we need to init our repository each time if we want to execute the build process.

Initializing local builds

Docker is all nice and practical, but docker comes with drawbacks:

- Inefficient handling of file storage
- Impractical file access (must do it through container)
- Risk of data loss due to running docker cleanup

Running the build process locally solves these issues.

Local init script

- Create a script **`scripts/install-sdk`**
- Place docker file contents inside it
- Docker image template
- Install script result

In order to have a reproducible process for initializing a new development machine we create a separate script that duplicates the same process as the one we do during docker image creation. We simply take the docker file and convert it into a runable shell script while removing any commands that are not relevant for local system install.

Running this script brings our local system into the same state as the docker image and having this whole process scripted helps making sure that we can easily duplicate this process.

Testing install script

- Run this script nightly
- Use base ubuntu docker image to run it
- Do not use it for every build (too slow)

To guarantee that the install script works, we can configure the nightly build pipeline to use basic ubuntu docker image and use the script to prepare that system for the build process. Then we execute the other scripts that are part of the build process and if this succeeds then the installation script was indeed successful in installing all necessary dependencies.

Compliance checking

- Runs several patch checks
- Verifies codeowners
- Reuses Zephyr check-compliance.py
- check_compliance.py script

In addition to the above checks, we also want to add zephyr compliance checking to our repository. This is implemented using zephyr/scripts/ci/check_compliance.py script. This script executes more checks for internal files in our repository. All we have to do is copy it to our repository and make a few changes to it:

Integration

```
$ mkdir -p scripts/ci/
$ cp ../zephyr/scripts/ci/check_compliance.py scripts/ci/
$ cp -r ../zephyr/scripts/zephyr_module.py scripts/
$ cp -r ../zephyr/scripts/kconfig scripts/
```

We also need to modify checkpatch and check_compliance a little.

To checkpatch we need to add a default root to our current directory:

```
    --- scripts/checkpatch.pl       2022-06-15 14:50:43.380589044 +0200
    +++ ../zephyr/scripts/checkpatch.pl     2022-06-13 15:34:30.741950201 +0200
    @@ -340,8 +340,6 @@
                                    } elsif ($0 =~ m@(.*)/scripts/[^/]*$@ &&

top_of_kernel_tree($1)) {
                                                        $root = $1;
    +               } else {
    +                       $root = "."
                                    }
                    }
```

To check compliance we need to make a change that replaces ZEPHYR_BASE since we are running it on our own code:

```
    --- scripts/ci/check_compliance.py      2022-06-15 14:51:10.632587148 +0200
    +++ ../zephyr/scripts/ci/check_compliance.py    2022-06-13 15:34:30.741950201
+0200
    @@ -25,8 +25,9 @@

    logger = None

    +# We set this to our code directory so that we run checks on local tree
    +ZEPHYR_BASE = os.path.realpath(os.path.dirname(os.path.realpath(__file__)) +
"/../../")
    -# This ends up as None when we're not running in a Zephyr tree
    -ZEPHYR_BASE = os.environ.get('ZEPHYR_BASE')
```

We also need to add a "dummy" Kconfig file to tests (or samples) because compliance checking checks these files and if it doesn't find any files at all then it fails:

```
menuconfig PLACEHOLDER
    bool "Dummy option"
    help
        This is a dummy option to make tests pass
```

After making these changes our check_compliance.py script should work:

```
$ ./scripts/ci/check_compliance.py
```

We should now add this to our pipeline. But in order to do that, let’s add a "check" script where we run check compliance and then add that check script to our pipeline. The benefit of doing this is that we can then later extend this shell script and reuse it between local machine and CI pipeline:

```
#!/bin/bash
# ./scripts/check

set -e

./scripts/ci/check_compliance.py || {
        echo "Checking compliance failed"
        exit 1
}
```

Codeowners sorting

- Ensures duplicates can be found and removed
- Easy to navigate the file

Another check we can add is related to codeowners file. This file often has many directory references and it is very convenient to keep it sorted.

To do this we add a shell script as a helper (sure, we could do this in python - but shell script works better in this case):

Sorting script

scripts/check-file-sorted

```
#!/bin/bash

# check that file is sorted
diff <(cat $1 | sort) <(cat $1) || {
    msg_err "File $1 is not sorted";
    echo "Run: sort -u -o $1 $1"
    exit 1
}
```

And the corresponding robot framework keyword looks like this:

```
Codeowners file is sorted
        ${result}=  Run Process  ${ROOT_DIR}/scripts/check-file-sorted
${CURDIR}/../../CODEOWNERS
        IF  ${result.rc} != 0
                Log To Console  CODEOWNERS file is not sorted
                Log To Console  ${result.stdout}
                Fail
        END
```

You can of course add the check directly to **`scripts/check`** as well. Where you place your checks is a matter of preference as long as they are reasonably grouped.

Merge request titles

- Possible to add links to ticket to the title
- Improves history navigation

It may sometimes be a good idea to check merge request title. In particular, if you are working on a private repository where commits never leave the repo, it is useful to have ticket numbers in the title because these are automatically converted to links making it possible to click on a commit title and go directly to the ticket it is related to.

We can make this either part of our integrity checks or part of a gitlint configuration. Here is how it can be added to integrity checks:

Script

```
#!/usr/bin/env python3
import os
import re

def check_merge_request_title():
  title = os.getenv("CI_MERGE_REQUEST_TITLE")
  regex = r"^Issue \#[0-9]+: [a-zA-Z0-9_\-\s]+$"
  # Check if title is present and does not match regex
  if(title and not re.match(regex, title)):
    raise BaseException("Merge request title (%s) does not match format (%s)" %
(title, regex))
```

We only check for regular expression match if the title is present. We don't want this check to fail locally - so it will only execute when our CI pipeline runs the robot framework checks on the repository.

```
Git merge request title is valid
    Check merge request title
```

The biggest problem with linking to issues from git history is that commits can migrate between repos. For example, a commit may be cherry picked into another repository, or the whole history may be pushed to a different service such as github. In such a scenario, the old issues are no longer present and what's even more inconvenient, the existing links may start pointing to completely unrelated tickets. So be aware of this!

Another commonly used approach is to prefix commit title with a subsystem that the commit is related to. The corresponding check would look similar to the above.

Robot framework script

- Use robot framework to maintain repo integrity
- Encode important decisions as part of integrity check
- Integrity checking script

If you want to encode these things into a robot framework script, here is a small example of what this may look like.

```
*** Settings ***
Library  OperatingSystem
Library  gitlab-ci.py

*** Test Cases ***

Gitlab ci has been setup
    Gitlab task template has been added
    Gitlab CI has a fully configured trunk pipeline
    Gitlab CI has a fully configured merge request pipeline
    Gitlab repository has merge method set to fast forward merge
    Gitlab repository has mandatory squashing
    Gitlab repository has merge checks enabled
    Gitlab repository has squash commit template
    Gitlab merge request template has been added

*** Keywords ***

Gitlab task template has been added
    File Should Exist  ${CURDIR}/../../.gitlab/issue_templates/task.md

Gitlab merge request template has been added
    File Should Exist  ${CURDIR}/../../.gitlab/merge_request_templates/default.md

Gitlab CI has a fully configured trunk pipeline
    Trunk pipeline runs robot tests

Gitlab CI has a fully configured merge request pipeline
    Merge request pipeline runs robot tests

Gitlab repository has merge method set to fast forward merge
    File Should Exist  ${CURDIR}/gitlab-ci/fast-forward-merge.jpg

Gitlab repository has mandatory squashing
    File Should Exist  ${CURDIR}/gitlab-ci/mandatory-squashing.jpg

Gitlab repository has squash commit template
    File Should Exist  ${CURDIR}/gitlab-ci/squash-commit-template.jpg

Gitlab repository has merge checks enabled
    File Should Exist  ${CURDIR}/gitlab-ci/merge-checks.jpg
```

There is a python extension we have added as well:

```
#!/usr/bin/env python3
import os
import yaml
```

```
PROJECT_ROOT = os.path.realpath(os.path.dirname(os.path.realpath(__file__)) +
"/../../")

def trunk_pipeline_runs_robot_tests():
  with open(PROJECT_ROOT + "/.gitlab-ci.yml", "r") as stream:
    yml = yaml.safe_load(stream);
    # We satisfy this by extending the pipeline
    if("merge_request_pipeline" not in yml["trunk_pipeline"]["extends"]):
      raise BaseException("Merge request pipeline should run 'robot scrum'")

def merge_request_pipeline_runs_robot_tests():
  with open(PROJECT_ROOT + "/.gitlab-ci.yml", "r") as stream:
    yml = yaml.safe_load(stream);
    # We satisfy this by running robot command on scrum folder
    if("robot scrum" not in yml["merge_request_pipeline"]["script"]):
      raise BaseException("Merge request pipeline should run 'robot scrum'")
```

This is a good example of how you can extend robot framework with python scripts. Here we can parse the gitlab-ci.yml file and check its content to verify that important settings are present.

You can make these checks as complex as you want. The key guiding principle is that if you reuse some checks across repositories then it benefits you to make them detailed.

We will use similar approach to integrity checking for other parts of our repository as well.

Pitfalls

- Not enough checks
- False successes
- Not scripting things outside of ci
- Zephyr CI pipeline.

The biggest pitfall when configuring CI pipeline is not adding enough checks or having checks that never run in CI (that time when you think you are checking something but you actually don’t and your pipeline reports a false successful build)

At this point in our training our pipeline is not very complex (and definitely not as complex as Zephyr github pipeline) - but it checks that EVERY task is completed.

As we add more code to our repository we want to make sure that absolutely every aspect of a merge request is properly checked. The pipeline will help us do this.

Success metrics

- Making important changes breaks CI
- No clutter in git history
- Fast flow of merge requests
- Trunk always in working condition

When you make important changes to the code or the repository and nothing breaks in CI, you should be weary. This means that CI is not working properly. When you make code changes without making corresponding changes to tests or other compliance checking code, the CI pipeline should break. This is like double entry ledger accounting: you make a change to code - the balance is off. You now need to make corresponding change to tests to "confirm" that you in fact meant to do it.

Your CI pipeline works to the degree that it can catch unintentional changes and pass only working code through into main trunk.

Your git history should look clean and should have a straight history. There should be no "Fix" commits in the main trunk.

Developers should be able to complete work quickly, get it reviewed by their peers, have the CI execute all checks and then merge it in a steady heartbeat of stable changes going straight into the trunk.

Finally, the main trunk version should be in excellent working shape. There can be work in progress in it - but it should be either fully working work in progress or a partial feature that is "invisible" to the user and enabled fully only after it has been completed.

What we covered

In this training we covered several important steps:

- **Installing dependencies** - and doing it in such a way that works on gitlab CI and not just on a local machine.
- **Setting up automatic merge request pipelines** - so that our merge requests are checked before they are merged into main.
- **Configuring merge requests** - so that our git history is kept clean and git commit messages are checked for compliance.
- **Zephyr compliance checking** - so that you can leverage the same compliance checking tests for your Kconfigs as zephyr uses.

We have also added all the checks to our CI pipeline so that it runs automatically.

We have two shell scripts and our robot scripts that are usable on a local machine for checking our repository:

- **scripts/init**: to initialize the repository
- **scripts/check**: to run compliance checks
- **robot scrum**: to run all scrum checks

Getting help

- Join weekly live: https://swedishembedded.com/live
- Use the SDK: https://github.com/swedishembedded/sdk

You can use the Swedish Embedded Platform SDK to quickly add the concepts described in this training to your firmware development workflow.

If you have any questions or need additional help, join the live sessions.

If you think this was a little too complicated for you and you would rather have somebody else set this up for you then book a call and let's discuss. You can book a call at https://swedishembedded.com/book-call.

```
Author: Martin Schröder, 15 Jun 2022
Email: martin.schroder@swedishembedded.com
Training: Zephyr Getting Started Training
Module: 2
```

Chapter 3

Directory Structure

► https://www.youtube.com/watch?v=Gl0U-HFmzCM *(YouTube video)*

What we have so far

- Base repository
- CI configured and working
- Basic compliance checking and patch checking
- Merge request workflow with templates

In the previous modules we have setup our repository, we have configured our CI, we have configured our compliance checking for some of the content in our repository.

Now it is time to make sure that we have a verified directory structure in place so that when we start adding source code to our repository, CI will enforce structure.

Problems we face

- No enforced directory structure
- No integrity checking for source directories

At this stage, if we were to add source code to our repository, we would not get any error from CI because we don't have any integrity checking in place that verifies directory structure. We now need to put in place integrity checking for a full directory structure of a typical embedded project.

The primary problem that we want to avoid later in the project is ending up with a directory structure that is difficult to manage and having to move things around. This disrupts any merge requests that we have in the review pipeline and it requires synchronization of the whole team - meaning that work has to momentarily stop to do the change.

What we want to do instead is:

- **Setup a proper directory structure** - so that all subsequent work fits somewhere within this structure.
- **Seamless merge requests** - we want to structure our files so that we can keep merge requests independent (this is important for team efficiency because it means we can merge in any order without having to worry about dependencies and file conflicts)
- **Clean organization** - we want to have all files in the project neatly organized so that we know where to find what.
- **Replicating Zephyr** - we want to replicate the zephyr directory structure as much as possible for our own files to minimize amount of things our developers need to keep in mind when working with our files.

What we will do

- Map out the whole directory structure
- Setup rules to verify directory structure consistency

In this module we will add the bulk of the remaining directory structure. We will add:

- **Application** - an application framework which will allow us later to add code to it and a template checked through robot framework which we can use to add more applications later.
- **Custom board** - that will serve as basis for adding our own boards that we can simulate in renode.
- **Driver template** - a template for drivers which we can use later to add more custom drivers to our project.
- **Libraries** - a directory where we can add libraries shared between our drivers and applications.
- **Samples** - that we will use for running robot framework tests in simulation. These will be used for system tests later (when we get to adding testing infrastructure itself).
- **CI Checks** - we will also add lots of new CI checks that will cover the newly added files from multiple angles - making sure that we only commit fully checked content.

We also need to map out where each kind of file is going to go. This means putting together a set of rules about the directory structure of our repository. For example:

- All public include files shall be placed under include directory and if they belong to a module they should be in corresponding subdirectory named after the module.
- Each public include file must have a corresponding C file with the same name.
- Drivers shall be placed under driver directory and each driver must have a test application.
- etc…

The benefit of this is that as the project progresses we will always be reminded about keeping the directory structure clean and consistent.

Directory tree overview

We need to have a place for:

- Applications
- Custom board definitions
- Cmake extensions
- Device drivers
- Documentation
- Device tree bindings
- Include files and sources
- Samples
- Tests

Zephyr already has a directory structure which you can read more about it in the Application Development section of the documentation.

What is less clear is how do you actually apply this to your own application and make it work. Zephyr doesn't cover that in much detail. So let's look at how the directory tree will work within our own project that we are basing on zephyr:

- **apps**. This directory will contain final firmware applications that you will be shipping to customers. You are not limited in the number of applications you can place here. Each application has it's own build configuration and CMake files.
- **boards**. Here we will be placing custom board definitions. We build applications for any number of boards - so boards are our "platforms" for which we are building our applications. A board definition consists of device tree and basic configuration for the board - as well as instructions for simulating and flashing firmware on the particular board.
- **cmake**. This directory will contain CMake scripts that act as libraries of CMake functions (".cmake" extension). These will be globally available to all components.
- **drivers**. Under this directory we will place hardware support - the drivers that plug into the zephyr driver model.
- **doc**. Here we will place the Zephyr documentation generator which we will adopt to our own needs to generate single PDF file with all doxygen docs combined with manually written technical documentation for our code.
- **dts**. This directory contains device tree bindings and shared device tree include files that need to be available to multiple platforms (just like "cmake" directory contains cmake files which are shared - this directory will contain ".dtsi" and "yaml" files for dts bindings).
- **include**. This will contain public headers which are visible to the whole project. We will try to keep these to a minimum (private headers go along side of code). These are shared across the whole project.
- **lib**. This tree will contain interfaces to third party libraries and shared project libraries as well. The vast majority of our project specific code will be organized into libraries which are shared across applications. The rest will be inside drivers and external modules.
- **samples**. Here we will be placing samples that we can run manually or inside a simulator to verify that our components work. Samples are different from tests in that they are interactive applications while tests are designed to be run as batch programs.

- **scripts**. Our build system infrastructure goes under scripts. Here we place our CI scripts, our compliance checking and all scripted processes within our repository.
- **tests**. This directory will contain our unit and integration tests. These are standalone applications that we build for native_posix platform and for actual hardware, which help us verify that our components work as expected in all scenarios.

This list contains the most basic directories that we will be adding in this training. If you are wondering where to place something then feel free to ask in the Swedish Embedded Discord Community.

Applications folders

- Contain your application (firmwares)
- You can have multiple applications in the same repo
- Applications can be placed outside of repo as well and linked with west

You can choose whether to place applications within the same directory as your main infrastructure or to place them outside as separate standalone projects. Zephyr and west are very flexible and allow both. For the remainder of this section we assume you are placing your applications under **`apps`**.

Application directory structure

The application folder consists of a tree of applications. Each application needs the following set of files:

Application directory structure

```
apps/shell/
├─── CMakeLists.txt ❶
├─── prj.conf ❷
├─── README.md ❸
├─── sample.yaml ❹
└─── src
    └─── main.c ❺
```

❶ Application CMake project file.
❷ Project configuration (Defaults for Kconfig)
❸ Application README
❹ Twister script that you can use to build the application.
❺ Application source code

Before we continue, we also have to add a RobotFramework script that will keep track of important details within this directory structure. We would like to have an automatic CI check that verifies that:

- **Existence of all files** - each application has the minimum files above that have to be present. We must check this.
- **Readme has all necessary sections** - so we don't forget anything (we may have multiple applications and we need to make sure that they all follow the same format).
- **Application name is correct** - this needs to be checked both in "CMakeLists.txt" file and in "sample.yaml" file as well.
- **Place for other application specific checks** - our RobotFramework script will be per application. As we develop our application, we later want to add checks to it that correspond to features of the application that have been completed.

Since both samples and all applications follow roughly the same structure, we will share our checks for the structure between multiple robot files. We will define these checks as a separate keyword.

Note

You need to make sure that both you and your fellow developers work with scrum scripts along side of all other tasks. It is through scrum scripts that we can mark a task as done. If work is being done outside of these checks then you end up with work in the repository

that is not checked by the scripts and such work is a breeding ground for broken functionality and other kinds of errors. So make sure that everyone is well versed in the concepts of DevOps on your team.

Let's ensure all of our applications follow the same template:

scrum/infrastructure/directory-structure.robot

```
*** Settings ***
Library  OperatingSystem

*** Test Cases ***

Apps have proper directory structure
        @{APPS} =       List Directories In Directory   ${CURDIR}/../../apps/
        FOR  ${APP}  IN  @{APPS}
                Set Test Variable  ${APP_NAME}  apps/${APP}
                Set Test Variable  ${APP_SOURCE}  ${CURDIR}/../../${APP_NAME}
                App directory has proper structure
        END

*** Keywords ***

App directory has proper structure
        File Should Exist  ${APP_SOURCE}/CMakeLists.txt
        File Should Exist  ${APP_SOURCE}/sample.yaml
        File Should Exist  ${APP_SOURCE}/prj.conf
        File Should Exist  ${APP_SOURCE}/README.md
        File Should Exist  ${APP_SOURCE}/src/main.c
```

Once we add this test then all of our directories under "apps" directory will be enforced to follow this structure automatically.

I have added a simple application to the apps folder as part of this merge request which fires up a shell. You can compile and run it like this:

```
$ west build -b native_posix apps/shell -t run
UART_0 connected to pseudotty: /dev/pts/71
*** Booting Zephyr OS build zephyr-v3.0.0  ***
```

We are not done yet, however. We also need to add checks that verify our YAML file. This is because we want to use twister to build all of our applications automatically and the data in the YAML file determines the build directories of these apps. We want to make sure that build directories follow the same pattern as well.

Let's add a check for our example application app folder:

scrum/apps/shell.robot

```
*** Settings ***
Library  ${CURDIR}/../infrastructure/directory-structure.py

*** Variables ***
${APP}  shell
${APP_NAME}  apps/${APP}

*** Test Cases ***

Application should have correct structure
```

```
        App title should be  Example shell application
        App should have release build
        App check integration platform  native_posix
```

In order to implement this check we also need to add a few methods in Python that help us verify the YAML file. So let's go ahead and add these as well:

scrum/infrastructure/directory-structure.py

```
#!/usr/bin/env python3
import os
import yaml
from robot.libraries.BuiltIn import BuiltIn

PROJECT_ROOT = os.path.realpath(os.path.dirname(os.path.realpath(__file__)) +
"/../../")

def get_app_name():
  APP = BuiltIn().get_variable_value("${APP_NAME}")
  if(not APP):
    raise BaseException("APP_NAME variable not defined!")
  return APP

def app_title_should_be(title):
  app = get_app_name()
  yaml_file = PROJECT_ROOT + "/" + app + "/sample.yaml"
  with open(PROJECT_ROOT + "/" + app + "/sample.yaml", "r") as stream:
    yml = yaml.safe_load(stream);
    # We satisfy this by extending the pipeline
    if("sample" not in yml or "name" not in yml["sample"]):
      raise BaseException("YAML file (%s) needs to have name of the sample" %
(yaml_file))
    if(yml["sample"]["name"] != title):
      raise BaseException("Application title for app '%s' must be '%s'" % (app,
title))

def app_should_have_release_build():
  app = get_app_name()
  yaml_file = PROJECT_ROOT + "/" + app + "/sample.yaml"
  with open(PROJECT_ROOT + "/" + app + "/sample.yaml", "r") as stream:
    yml = yaml.safe_load(stream);
    # We satisfy this by extending the pipeline
    test_id = app.replace("/", ".") + ".release"
    if("tests" not in yml or test_id not in yml["tests"]):
      raise BaseException("Application '%s' does not have a release build (%s)
defined in sample.yaml!" % (app, test_id))

def app_check_integration_platform(platform):
  app = get_app_name()
  yaml_file = PROJECT_ROOT + "/" + app + "/sample.yaml"
  with open(PROJECT_ROOT + "/" + app + "/sample.yaml", "r") as stream:
    yml = yaml.safe_load(stream);
    # We satisfy this by extending the pipeline
    test_id = app.replace("/", ".") + ".release"
    if("tests" not in yml or test_id not in yml["tests"]):
      raise BaseException("Application '%s' does not have a release build (%s)
defined in sample.yaml!" % (app, test_id))
```

```
    if(platform not in yml["tests"][test_id]["integration_platforms"]):
      raise BaseException("Application '%s' does not have '%s' among integration
platforms." % (app, platform))
```

The above checks accomplish a few important things:

- **Standardized build paths** - we make sure that when we build our applications with Twister (test automation tool we will use) they will have fairly standard paths making them easy to find.
- **Scrum check for all tasks related to an application** - we have a separate scrum check in addition to the generic directory structure check which checks details about the application - we can potentially extend this check to include many other tasks.

At this point we would like to build our shell application so let's add a build script that will build apps, tests and samples:

scripts/build

```
#!/bin/bash

ROOT="$(realpath $(dirname $BASH_SOURCE)/..)"

set -e

$ROOT/../zephyr/scripts/twister \
        --integration \
        -c \
        -O build-all \
        -v \
        -i \
        -A boards \
        -T apps \
        -T tests \
        -T samples || {
        echo "Error while trying to build project"
}
```

When we run this script, all of our build artifacts will be neatly organized by twister under build-all folder by target platform:

```
build-all/
├──── native_posix ❸
│    ├──── shell
│    │    └──── apps.shell.release ❶
│    │         ├──── app
│    │         ├──── build.log
│    │         ├──── CMakeCache.txt
│    │         ├──── CMakeFiles
│    │         ├──── cmake_install.cmake
│    │         ├──── compile_commands.json
│    │         ├──── Kconfig
│    │         ├──── Makefile
│    │         ├──── modules
│    │         ├──── zephyr ❷
│    │         ├──── zephyr_modules.txt
│    │         └──── zephyr_settings.txt
```

❶ application build folder with standard name

2. binaries are here (zephyr.elf)
3. platform specific build folder (from integration_platforms)

This gives us a powerful way to build all applications within our project. From now on, we don't need to worry about the build process very much. We can still build applications manually, but the CI process will build everything in one go making sure that all of our applications compile (and some run - such as tests) properly.

Custom boards

- Custom boards define custom build targets
- You can extend zephyr boards
- You can build for multiple boards at once

The next thing we are going to do is add a custom board to our project. This is important if you are developing your own hardware because you will always have some form of custom configuration for the hardware you are using.

Zephyr supports this out of the box using device tree which is used for configuring your hardware. For the sake of this training we will simply copy an existing zephyr board into our project so I can show you how this is done. If you need help you should join our monthly mastermind. Start by getting in touch: here.

Structure

- CMake files
- Kconfig settings
- Defconfig defaults
- Device tree
- Board definition (yml)
- JTAG configuration

Tests we need for our boards folder:

scrum/infrastructure/directory-structure.robot

```
Boards have proper directory structure
        @{ARCHS} =      List Directories In Directory   ${CURDIR}/../../boards/
        FOR  ${ARCH}  IN  @{ARCHS}
                @{BOARDS} =     List Directories In Directory
${CURDIR}/../../boards/${ARCH}/
                FOR  ${BOARD}  IN  @{BOARDS}
                        Set Test Variable  ${ARCH}  ${ARCH}
                        Set Test Variable  ${BOARD}  ${BOARD}
                        Set Test Variable  ${BOARD_SOURCE}
${CURDIR}/../../boards/${ARCH}/${BOARD}
                        Board directory has proper structure
                END
        END

*** Keywords ***

Board directory has proper structure
        File Should Exist  ${BOARD_SOURCE}/board.cmake (1)
        File Should Exist  ${BOARD_SOURCE}/Kconfig.board (2)
        File Should Exist  ${BOARD_SOURCE}/Kconfig.defconfig (3)
        File Should Exist  ${BOARD_SOURCE}/${BOARD}_defconfig (4)
        File Should Exist  ${BOARD_SOURCE}/${BOARD}.dts (5)
        File Should Exist  ${BOARD_SOURCE}/${BOARD}.yaml (6)
        File Should Exist  ${BOARD_SOURCE}/support/openocd.cfg (7)
```

(1) Board compilation flags and CMake includes

❷ Board Kconfig options (including what this board depends on)
❸ Board Kconfig defaults (sets defaults using Kconfig syntax)
❹ Defaults set as raw options (only enable bare minimum)
❺ Board device tree definition (all devices are configured here)
❻ Board yaml definition - used by twister to select board as target platform
❼ OpenOCD settings for flashing firmware to this board (west run command uses these)

We can copy the nucleo_f401re board from zephyr and use it as an example:

```
$ mkdir -p boards/arm
$ cp -r ../zephyr/boards/arm/nucleo_f401re boards/arm/custom/
```

For now we don't care about being able to simulate our application on renode - we will add that later when we add comprehensive testing. We just want to be able to build our application for our newly added board:

In order to use our new board we need to rename it to a custom name that does not conflict with the Zephyr board that has the same name. So let's rename it to "custom_board":

```
mv boards/arm/nucleo_f401re/ boards/arm/custom_board
mv boards/arm/custom_board/nucleo_f401re_defconfig
boards/arm/custom_board/custom_board_defconfig
mv boards/arm/custom_board/nucleo_f401re.dts boards/arm/custom_board/custom_board.dts
mv boards/arm/custom_board/nucleo_f401re.yaml
boards/arm/custom_board/custom_board.yaml
```

Make sure you also modify "identifier" field in custom_board.yaml.

```
$ west west build -b custom_board apps/shell
```

We also need to add this platform to our application "sample.yaml" file:

```
tests:
  apps.shell.release:
    build_only: true
    integration_platforms:
      - native_posix
      - nucleo_f401re ❶
```

❶ add platform here

There is one final thing we need to do before we can use west (and twister) to build our application for our new board. We need to tell west about our board directory and that we have a board directory in our project. This is done by adding the following file:

zephyr/module.yml

```
build:
    kconfig: Kconfig ❸
    cmake: . ❹
  settings:
    board_root: . ❶
    dts_root: . ❷
```

❶ Specify that we have a local boards directory
❷ Specify that we have a local dts root
❸ This will ensure west includes our top level Kconfig
❹ This will ensure west includes our top level CMakeLists.txt file

We can now use our build script to build the app:

```
$ ./scripts/build
```

You should get the following result:

```
INFO    - 1 test scenarios (2 configurations) selected, 0 configurations discarded
due to filters.
INFO    - Adding tasks to the queue...
INFO    - Added initial list of jobs to queue
INFO    - 1/2 native_posix              shell/apps.shell.release
PASSED (build)
INFO    - 2/2 custom_board              shell/apps.shell.release
PASSED (build)
```

We now have ability to build our application for our custom board.

CMake Extensions

- Placed under **`cmake`** directory
- Included from top level CMake file
- Have access to the whole build process for each app
- Used to add cmake extensions to the build process

In this directory we will just add an empty CMakeLists.txt file which we will use later to add more features to our project.

cmake/CMakeLists.txt

```
# Empty for now
```

We also need to include this file in the top level CMakeLists.txt file:

CMakeLists.txt

```
add_subdirectory(cmake)
```

That's all we are going to be adding for now.

Device drivers

- Special source files that plug into generic device interfaces
- Instantiated automatically by device tree code
- Implement link between generic interface and hardware device

Next we are going to add an example driver. We will check that everything is working by building and running our test application.

To add a driver we need to add a drivers directory and then place our code in there.

In Zephyr, the "drivers" directory is split into driver types (i.e. watchdog, timer, etc.). We will follow the same structure.

Since "drivers" directory is just like any other source directory, we need to add both CMakeLists.txt file into each subdirectory and also a Kconfig file that adds menuconfig options for our driver.

Here is a test we can add to make sure that we don't forget any files:

scrum/infrastructure/directory-structure.robot

```
Drivers have proper structure
        File Should Exist  ${ROOT_DIR}/drivers/CMakeLists.txt
        File Should Exist  ${ROOT_DIR}/drivers/Kconfig
        @{DRIVER_TYPES} =       List Directories In Directory   ${ROOT_DIR}/drivers/
        FOR  ${DRIVER_TYPE}  IN  @{DRIVER_TYPES}
                File Should Exist  ${ROOT_DIR}/drivers/${DRIVER_TYPE}/CMakeLists.txt
                File Should Exist  ${ROOT_DIR}/drivers/${DRIVER_TYPE}/Kconfig
                @{DRIVERS} =    List Directories In Directory
${ROOT_DIR}/drivers/${DRIVER_TYPE}/
                FOR  ${DRIVER}  IN  @{DRIVERS}
                        # Any per driver checks will go here
                END
        END
```

Now let's add a CMakeLists.txt file for our driver:

drivers/example/CMakeLists.txt

```
zephyr_library()

zephyr_library_sources_ifdef(CONFIG_EXAMPLE_DRIVER example_driver.c)
```

I have chosen to make a library out of each driver type. You can decide how you want to do this. You may include the sources into the main zephyr build without creating a library. This is just one way of organizing the sources that works quite well.

For our "Kconfig" file we need to add config options for our new driver:

drivers/example/Kconfig

```
config EXAMPLE_DRIVER
    bool "Example driver"
    help
      This is our example driver.
```

Next we need to "tie in" our CMake file and Kconfig into the build system. We do this by creating a few other files:

drivers/CMakeLists.txt

```
# Add driver type directories here

add_subdirectory(example)
```

drivers/Kconfig

```
menu "Device Drivers"

rsource "example/Kconfig"

endmenu
```

These options will appear under "Modules > zephyr-getting-started > Device Drivers" in menuconfig:

```
$ west build -b custom_board apps/shell -t menuconfig
```

NOTE

> if you want to place them at top level then you can do so as well by directly including the top level Kconfig file from within your application Kconfig.

We also need to include these files from the top level Kconfig and CMakeLists.txt:

CMakeLists.txt

```
add_subdirectory(drivers)
```

Kconfig

```
rsource "drivers/Kconfig"
```

Now let's add the source code for our driver:

drivers/example/example_driver.c

```
/** Device tree compatible */
#define DT_DRV_COMPAT example_driver ❶

#include <errno.h>
#include <device.h>
#include <logging/log.h>

/** Create a logger */
LOG_MODULE_REGISTER(example_driver); ❷

/** Driver local data */
struct example_driver {
        // data goes here
};

/** Driver configuration (from device tree) */
struct example_driver_config {
        // config variables
};

/** Driver initialization */
static int _example_driver_init(const struct device *dev){
        if(!dev || !dev->data || !dev->config)
                return -EINVAL;
        LOG_INF("Example driver initialized!");
        return 0;
```

```
}

/** Macro to instantiate the driver */
#define EXAMPLE_DRIVER_INIT(n) \
        static struct example_driver _example_##n; \
        static const struct example_driver_config _example_config_##n = {\
        }; \
        DEVICE_DT_INST_DEFINE(n, _example_driver_init, NULL, &_example_##n,
&_example_config_##n, POST_KERNEL, CONFIG_APPLICATION_INIT_PRIORITY, NULL)

/** Instantiate the driver */
DT_INST_FOREACH_STATUS_OKAY(EXAMPLE_DRIVER_INIT); ❸
```

❶ this will make a device tree node compatible with our driver
❷ register a logger for this driver (not strictly necessary)
❸ instantiate the driver data (statically instantiated)

Before we can actually build our application, we need to create a device tree binding and add a device tree node that will be assigned an instance of driver data.

In our device tree we add a node like this:

boards/arm/custom_board/custom_board.dts

```
/ {
....
    example_driver {
        compatible = "example-driver";
        status = "okay";
    };
....
};
```

Or if we would like to enable our driver for a native posix build then we need to add a device tree overlay since the native posix dts file is defined within zephyr (we don't have direct access to it). Zephyr supports overlays to allow us to add extra devices to existing platform builds.

apps/shell/boards/native_posix.overlay

```
/ {
    example_driver {
        compatible = "example-driver";
        status = "okay";
    };
};
```

If we now build and run our application then we will see text printed out in the console:

```
$ west build -b native_posix apps/shell -t run
UART_0 connected to pseudotty: /dev/pts/71
Example driver initialized! ❶
*** Booting Zephyr OS build zephyr-v3.0.0  ***
```

❶ Our driver initializes before application runs

Documentation

- We will place it under **doc** folder
- Has separate build process
- We produce PDF and HTML versions

Zephyr comes with an existing documentation generator. This generator is based on "sphinx" and "breathe" tools that enable you to write your docs in reStructuredText and combine them with Doxygen documentation. This is a very powerful system, but requires a separate module to cover in full. We will do this in a later module.

For now, we just want to create an empty folder for our documentation.

```
$ touch doc/.empty
```

Device tree bindings

- Connect device tree to C code through generated headers
- Define a schema for device tree nodes
- Required for each device tree node
- Assigned based on "compatible" string

We have added an example driver to our application, but we still don't have a way to get data from the device tree node to the application code. To do this we need to add device tree bindings. Zephyr supports device tree bindings written in YAML format.

We add the device tree bindings under "dts/bindings/" directory.

Let's add a simple required variable to our example driver:

dts/bindings/example-driver.yaml

```
description: |
    Device tree bindings for the example driver.

    Example definition in device tree:

    example_driver {
        compatible = "example-driver";
        custom-variable = <1>;
        status = "okay";
    };

compatible: "example-driver"

include: base.yaml

properties:
  custom-variable:
    type: int
    required: true
    description: A custom variable
```

If we now try to build our application then the build process will fail because we still haven't added the required variable to our device tree definition.

Once we have added the variable to device tree, we can retrieve it in our code by using DT_INST_PROP(n, custom_variable) macro.

```
....
    .custom_variable = DT_INST_PROP(n, custom_variable)
....
```

This goes into the initialization of our "struct example_driver_config" structure.

Include files

- Holds public interface headers
- Usually globally visible

We place include files under **`include`** directory. It's important to note that this directory should only contain "public" interface headers. Any private headers should be placed alongside of source files in the directory of the source files that need them.

The include directory is currently empty. Add an ".empty" file to it and then add it to top level CMakeLists.txt as an include path. This will allow you to reference files inside this directory anywhere in your project:

CMakeLists.txt

```
zephyr_include_directories(include)
```

Libraries and shared code

There are two ways to add libraries to our project:

- Placing them into a separate **`lib`** directory
- Adding them as zephyr west submodules

In this section we will simply look at how we can organize shared code within the same repository as our other projects. We will place them under **`lib`** directory tree.

The lib directory follows the same structure as drivers, except that each subdirectory under lib represents a library. A library is nothing more than a collection of files that simply have a common purpose. Posix implementation on zephyr is a library for example.

A library that we place under lib directory is not a standalone CMake project. Instead, lib directory only contains sources for our library. The public interface headers will go under root folder "include/<library>" directory so that they can be referenced in application sources as "#include <some_library/some_interface.h>"

The lib directory in your Zephyr project is the most generic way of sharing code between applications, drivers and other components. You would typically separate out the business logic of your application into a set of libraries.

If a library really exists as a separate standalone project then you would make it a west module instead. We will look at this a bit later when we add CMock and Unity libraries as external modules.

Understand that when we place something under "lib" folder, we expect the code to be formatted according to our custom format rules. This code also needs to be fully tested. In fact, we will add checks that will ensure that we can not add a single source file to any location in our project without also adding a corresponding test. This is one reason you may want to place code inside lib folder instead of placing it outside of the repository as a separate Zephyr module.

Let's add an example library to our project.

We start with a RobotFramework test that will check our folder structure:

scrum/infrastructure/directory-structure.robot

```
Libraries have proper structure
    File Should Exist  ${ROOT_DIR}/lib/CMakeLists.txt
    File Should Exist  ${ROOT_DIR}/lib/Kconfig
    @{LIBRARIES} =  List Directories In Directory   ${ROOT_DIR}/lib/
    FOR  ${LIBRARY}  IN  @{LIBRARIES}
        File Should Exist  ${ROOT_DIR}/lib/${LIBRARY}/CMakeLists.txt
        File Should Exist  ${ROOT_DIR}/lib/${LIBRARY}/Kconfig
        @{SOURCES} =    List Directories In Directory   ${ROOT_DIR}/lib/${LIBRARY}/
        FOR  ${SOURCE}  IN  @{SOURCES}
            # Any per driver checks will go here
            No Operation
        END
    END
```

We then add our library directory, "lib/example", where we place the source code, cmake file and Kconfig:

lib/example/Kconfig

```
config EXAMPLE_LIBRARY
    bool "Example library"
    help
        Example library
```

Then we add the cmake file that adds our library source to the build process:

lib/example/CMakeLists.txt

```
zephyr_library()
zephyr_library_sources_ifdef(CONFIG_EXAMPLE_LIBRARY example.c)
```

Then we bind these into the build process by making sure higher level files include them:

lib/Kconfig

```
rsource "example/Kconfig"
```

lib/CMakeLists.txt

```
add_subdirectory(example)
```

Also remember to add the lib folder to top level CMake and Kconfig:

CMakeLists.txt

```
....
add_subdirectory(lib)
....
```

Kconfig

```
....
rsource "lib/Kconfig"
....
```

Finally we add a public interface and a source file:

include/example/example.h

```
#pragma once

#include <stdint.h>

struct example_object { ❶
    uint8_t dummy;
};

/**
 * \brief Initializes example object.
 * \param[in] self reference to object
 * \returns error status
 * \retval -EINVAL invalid arguments
 * \retval 0 success
 **/
int example_object_init(struct example_object *self); ❷
```

❶ define our object
❷ define a simple constructor

lib/example/example.c

```
#include <memory.h>
#include <kernel.h>
```

```
#include <example/example.h>

int example_object_init(struct example_object *self){
       if(!self)
               return -EINVAL;
       memset(self, 0, sizeof(*self));
       printk("Example object initialized!\n");
       return 0;
}
```

As a simple example we can then use our library in our example application by adding it to the prj.conf:

apps/shell/prj.conf

```
CONFIG_EXAMPLE_LIBRARY=y
```

And then use the code:

apps/shell/src/main.c

```
struct example_object _ex; (1)
if (example_object_init(&_ex) != 0){
    printk("Could not initialize example object\n");
}
```

(1) allocate local instance of example_object

Sample applications

- Primary way of working with parts of the application in isolation.
- Similar to "demo" code but very useful during development.

There is no difference in how Zephyr treats samples from how it treats our applications. We have introduced a difference by adding an "apps" folder (which is not part of the zephyr tree) because we want to create a logical separation between what an "app" is (a releasable product) and what a sample is (a way to run usage scenarios and demonstrate concepts).

We want to use samples to test and demonstrate partial concepts - while ensuring that no unnecessary changes are done to our production apps. Therefore we keep them separate - but the folder structure is the same. Now you also know why there is a sample.yaml file in application folder (and not application.yaml) - it's because applications are treated on our level as zephyr treats samples. This allows us to reuse twister functionality that builds samples in order to build our applications.

I prefer to add a separate test that checks all samples which does similar things to what the apps test does:

scrum/infrastructure/directory-structure.robot

```
Samples have proper directory structure
    @{APPS} =       List Directories In Directory  ${ROOT_DIR}/samples/
    FOR  ${APP}  IN  @{APPS}
        Set Test Variable  ${APP_NAME}  samples/${APP}
        Set Test Variable  ${APP_SOURCE}  ${ROOT_DIR}/${APP_NAME}
        App directory has proper structure
    END
```

Since our sample is a deliverable, we should define a scrum task for this as well:

scrum/samples/example.robot

```
*** Settings ***
Library  ${CURDIR}/../infrastructure/directory-structure.py

*** Variables ***
${APP}  example
${APP_NAME}  samples/${APP}

*** Test Cases ***

Application should have correct structure
        App title should be  Example sample
        App should have release build
        App check integration platform  native_posix
```

Scripts and CI automation

- Entry point to all build tasks
- Should be organized under **`scripts`** directory tree

Scripts is the only folder under which we are allowed to have executable scripts. We actually have a compliance check for this which already runs as part of checking compliance. We use the scripts folder for all "scripting".

Let’s add a robot script that checks our scripts folder!

scrum/infrastructure/directory-structure.robot

```
Scripts have proper structure
    There is a repository init script
    There is a script that builds all configurations
    There is a script that checks compliance

There is a repository init script
    File Should Exist  ${ROOT_DIR}/scripts/init

There is a script that builds all configurations
    File Should Exist  ${ROOT_DIR}/scripts/build

There is a script that checks compliance
    File Should Exist  ${ROOT_DIR}/scripts/check
```

As an exercise you can try adding your own robot scripts that check your scripts folder.

Unit and integration tests

- Contains test applications
- Used for unit and integration test implementation

Lastly we are going to add a tests folder. This folder will contain unit and integration tests. As I described in the training on continuous testing, you need to have both unit and integration tests because they fulfill different purposes. We will revisit the topic of tests in a module dedicated to testing.

For now, we just want to add a test folder so that we have a place where we can later put our tests.

Normally we would not allow even pushing all of this code without achieving full code coverage. We will tighten the checks before this training is completed.

The tests directory will contain tests organized by type of test. I personally prefer not to have too many directory levels - but you can organize your tests in any way you like. Twister doesn't care about the directory structure so it's mostly there for your convenience.

Let's add a robot script that will check our tests:

scrum/infrastructure/directory-structure.robot

```
Tests have proper directory structure
    @{TEST_CLASSES} =       List Directories In Directory   ${ROOT_DIR}/tests/
    FOR  ${TEST_CLASS}  IN  @{TEST_CLASSES}
        @{TEST_TYPES} = List Directories In Directory
${ROOT_DIR}/tests/${TEST_CLASS}/
        FOR  ${TEST_TYPE}  IN  @{TEST_TYPES}
            @{TESTS} =      List Directories In Directory
${ROOT_DIR}/tests/${TEST_CLASS}/${TEST_TYPE}/
            FOR  ${TEST}  IN  @{TESTS}
                Set Test Variable  ${TEST_NAME}  ${TEST_CLASS}/${TEST_TYPE}/${TEST}/
                Set Test Variable  ${TEST_SOURCE}  ${ROOT_DIR}/tests/${TEST_NAME}/
                Test directory has proper structure
                # A test should test only one C file and that file should exist
                # Organize your directories such that your tests correspond to
                # the source files you are testing
                File Should Exist  ${ROOT_DIR}/${TEST_CLASS}/${TEST_TYPE}/${TEST}.c
            END
        END
    END
...
Test directory has proper structure
    File Should Exist  ${TEST_SOURCE}/CMakeLists.txt
    File Should Exist  ${TEST_SOURCE}/testcase.yaml
    File Should Exist  ${TEST_SOURCE}/prj.conf
    File Should Exist  ${TEST_SOURCE}/README.md
    File Should Exist  ${TEST_SOURCE}/src/unit.c
    File Should Exist  ${TEST_SOURCE}/src/integration.c
    Check Test Configuration
```

We implement the "Check Test Configuration" in python since we want to verify a few things.

- **That test has proper id** - that follows test path.

- **That test is tagged with the source filename** - so that we can later run only tests that have the source file tag of a particular file.

scrum/infrastructure/directory-structure.py

```
def check_test_configuration():
    TEST_NAME = BuiltIn().get_variable_value("${TEST_NAME}")
    yaml_file = PROJECT_ROOT + "/tests/" + TEST_NAME + "/testcase.yaml"
    with open(yaml_file, "r") as stream:
        yml = yaml.safe_load(stream);
        if("tests" not in yml):
            raise BaseException("Test '%s' does not have 'tests' section in
testcase.yaml file" % (TEST_NAME))
        # We satisfy this by extending the pipeline
        test_id = TEST_NAME.replace("/", ".") + "unit"
        source_file = TEST_NAME.strip("/") + ".c"
        if(test_id not in yml["tests"]):
            raise BaseException("Test '%s' does not have a unit test (%s) defined in
testcase.yaml!" % (TEST_NAME, test_id))
        if(source_file not in yml["tests"][test_id]["tags"]):
            raise BaseException("Test '%s' does not have tag '%s'" % (TEST_NAME,
source_file))

        test_id = TEST_NAME.replace("/", ".") + "integration"
        if(test_id not in yml["tests"]):
            raise BaseException("Test '%s' does not have an integration test (%s)
defined in testcase.yaml!" % (TEST_NAME, test_id))
        if(source_file not in yml["tests"][test_id]["tags"]):
            raise BaseException("Test '%s' does not have tag '%s'" % (TEST_NAME,
source_file))
```

This is sufficient for now.

Additional checks

- Code formatting
- CMake formatting
- Python formatting

Before we move on, there are a few additional checks that are good to implement right now because leaving it for later will result in having to reopen this task again.

These are:

- **Code formatting check** - we need to ensure that all of our newly added sources have correct formatting.
- **CMake formatting check** - we need to format our CMake files.
- **Python formatting check** - we need to check the formatting of Python code.
- **Extend compliance check range** - we currently have checkpatch checking each commit through the git hook but we also want to check the whole MR diff on CI and not just the last commit (which is the default). We can use commit range passed to compliance checking script for this.
- **Automatic formatting script** - that will make commit experience much less time consuming because we will be able to fix a lot of formatting issues automatically.

Let's first make sure that our compliance checking runs checkpatch and all other checks on the whole diff of the MR. For this we need to set commit range to cover diff between target branch and our current HEAD commit:

scripts/check

```
if [[ -v CI_MERGE_REQUEST_TARGET_BRANCH_NAME ]];
then
    # to get our checks working on ci it is necessary to fetch origin
    git fetch origin $CI_MERGE_REQUEST_TARGET_BRANCH_NAME
    COMMIT_RANGE="origin/$CI_MERGE_REQUEST_TARGET_BRANCH_NAME..HEAD"
else
    COMMIT_RANGE="HEAD~1..HEAD"
fi

./scripts/ci/check_compliance.py \
    -c $COMMIT_RANGE || {
    echo "Checking compliance failed"
    exit 1
}
```

TIP

You can also use "git rev-list HEAD | tail -n 1" to get the SHA of the first commit in your repository. Do not run checks on all commits by default though. Instead, you can add a flag to "check" script that will allow you to check all commits occasionally.

Next we want to check the formatting of the files.

For this we use:

- **clang-format** to check C and H files
- **cmake-format** to check cmake files
- **black** to check python files

We can use the same git commit range to find the files and run scripts on them. This looks like this:

scripts/check

```
check_file_formatting(){
    FILE=$1
    if [[ $FILE == *.py ]]; then
        black --check $FILE || {
            echo "Checking formatting of $FILE failed"
            echo "Run: ./scripts/fix"
            return 1
        }
    elif [[ $FILE == *.c ]] || [[ $FILE == *.h ]]; then
        clang-format --dry-run --style=file -Werror --fallback-style=none $FILE || {
            echo "Checking formatting of $FILE failed"
            echo "Run: ./scripts/fix"
            return 1
        }
    elif [[ $FILE == */CMakeLists.txt ]] || [[ $FILE == *.cmake ]]; then
        cmake-format --first-comment-is-literal --check $FILE || {
            echo "Checking formatting of $FILE failed"
            echo "Run: ./scripts/fix"
            return 1
        }
    fi
}

# Check file formatting for added and modified files
for FILE in `git diff --name-status $COMMIT_RANGE | grep "^[AM]" | cut -f 2`; do
    check_file_formatting $FILE || {
        exit 1
    }
done

# Check file formatting for renamed files
for FILE in `git diff --name-status $COMMIT_RANGE | grep "^[R]" | cut -f 3`; do
    check_file_formatting $FILE || {
        exit 1
    }
done
```

We have to use a different command when listing renamed files because git output for these files is slightly different.

This ensures we can run our checks only on files for which these checks are relevant so it will be faster.

As far as fixing errors automatically goes, we need to add a similar script but make it fix the errors in place.

scripts/fix

```
fix_file_formatting(){
    FILE=$1
    if [[ $FILE == *.py ]]; then
        black $FILE || {
```

```
            echo "Fixing formatting of $FILE failed"
            echo "You'll have to fix this manually"
            return 1
        }
    elif [[ $FILE == *.c ]] || [[ $FILE == *.h ]]; then
        clang-format -i --style=file -Werror --fallback-style=none $FILE || {
            echo "Fixing formatting of $FILE failed"
            echo "You'll have to fix this manually"
            return 1
        }
    elif [[ $FILE == */CMakeLists.txt ]] || [[ $FILE == *.cmake ]]; then
        cmake-format --first-comment-is-literal -i $FILE || {
            echo "Fixing formatting of $FILE failed"
            echo "You'll have to fix this manually"
            return 1
        }
    fi
}

# Check file formatting for added and modified files
for FILE in `git diff --name-status $COMMIT_RANGE | grep "^[AM]" | cut -f 2`; do
    fix_file_formatting $FILE || {
        exit 1
    }
done

# Check file formatting for renamed files
for FILE in `git diff --name-status $COMMIT_RANGE | grep "^[R]" | cut -f 3`; do
    fix_file_formatting $FILE || {
        exit 1
    }
done
```

The above fixes the files in place. If you want, you can also add checks to make sure you don't fix files unless they are staged or committed to git (to avoid possible loss of work). I'll leave that up to you.

Pitfalls

- Missed checks for script return values
- Diverging too much from Zephyr directory structure
- Insufficient integrity checking

The biggest pitfall at this point is **not adding enough checks**. It is important to tighten the checks so much that everything you commit is as well-checked as possible.

Also watch out for the following:

- **Missed return code checks** - if you miss checking return code of a shell script and exiting with an error, you may end up missing faults on CI. "set -e" does not work inside nested functions!
- **Missed checks** - thinking you checked something but you haven't. To identify this you can sometimes deliberately introduce errors and see if they are caught by checks.
- **Too many shared files** - if your directory structure has too many shared files then working collaboratively on the project will become a bottleneck because of frequent conflicts.
- **Not following zephyr structure** - if you try to invent your own structure you will put extra strain on developers which is undue. You should try to structure your application in very similar ways to zephyr - this will ensure you can reuse parts of Zephyr infrastructure as needed.

Success metrics

- Adding files in wrong places triggers appropriate CI checks
- Forgetting to add related files triggers ci checks
- Inconsistent code formatting triggers ci checks

By now you should have a fully functional directory structure where you can start placing your custom project files.

The main success metrics are as follows:

- **All C and H files are checked for formatting** - this should be done according to included ".clang-format" file in your project directory.
- **All cmake and python files are checked for formatting as well** - this should be done automatically through your check script that runs on CI as well.
- **Applications, samples and tests must be buildable** - using a single command. This should be done on CI as well.

What we covered

- File organization and placement
- Integrity checking of the directory structure

This was the longest module of this training so far and I hope that you have learned a lot.

We have covered:

- **Structure of the applications folder** - and how we add applications.
- **How to add custom board support** - and check that directories have correct structure.
- **Where our custom cmake libraries can be added** - so that we can later extend cmake with custom libraries.
- **How and where to add device drivers** - and how to automatically check the directory structure.
- **Where our documentation will be added** - the docs folder.
- **How to add device tree bindings** - so that you can have custom variables in your device tree.
- **Where you will place shared public interfaces** - the shared include directory for your project.
- **Where to place libraries** - the lib directory and how to check it.
- **Samples directory** - and the purpose of samples in your project.
- **Tests directory** - and how it differs from samples and applications.
- **Automatic checks** - so that your files are checked before a merge request is allowed to be merged.

Getting help

- Get consulting: https://swedishembedded.com/book-call

You can of course implement all of this yourself, but if your team would rather be doing something else then Swedish Embedded Consulting Group is here to help.

We can do the following for you:

- **Setup all infrastructure for Zephyr development** - so you don't have to fiddle with a bunch of bash and python scripts yourself.
- **Refactor and modernize legacy code** - so that you can focus on new development.
- **Training of your team in Zephyr development** - so you understand how to use the kernel to solve real life problems faster than you would solve them if you implement everything yourself.

If this sounds good then book a call with us. We have solved the issue of having to find a good time for a call by providing you with an automatic calendar where you can directly pick a time that works best for you.

Click here to book a FREE no-strings-attached call with us to discuss your firmware development situation

```
Author: Martin Schröder, 17 Jun 2022
Email: martin.schroder@swedishembedded.com
Training: Zephyr Getting Started Training
Module: 3
```

Chapter 4

Documentation Generation

► https://www.youtube.com/watch?v=Jd5WMwZfKcY *(YouTube video)*

Recap

- We have a repository
- We have automated CI
- We have a verified directory structure

At this point the repository is in good shape, but it is still only half way to being completed fully set up.

Problems we face

- We still don’t have an automated way to generate documentation
- We don’t have a system in place to pull documentation from source code

There is still no way to generate documentation from multiple sources (such as cmake files, rst files, sources and kconfig files) and to create pdf and html versions of this content in a scripted and automated fashion.

This means that the project doesn’t have documentation that can be dstributed to the users of the code - whether it is the team members or customers.

Without proper documentation, the project is very likely to decay in quality. The simple act of having developers go over their code and write "about it" (ie document what it does and how to use it) has the tendency to make developers look one extra time at the code and make it better.

Solution

- Reuse Zephyr documentation generation scripts
- Combine documentation from different parts of the repository in one place.
- Generate HTML and PDF file from the data

What we need is a documentation generation system.

In this module we are going to adopt existing (and very powerful) documentation generation process that is already being used in Zephyr to your firmware project.

Zephyr uses a combination of "sphinx" and custom scripts to generate both a PDF file with all documentation and an HTML website.

Documentation in Zephyr is generated from a combination of files written as text in RST format and data contained within the repository (for example the device tree bindings). The custom scripts parse this data from repository and use it to improve the docs.

Not only this, but we also have direct access to doxygen documentation so we can include it inline into our Zephyr docs.

In this module we will be looking at how we can adopt the zephyr documentation generator to our custom project. At the very minimum we will need to do the following:

- **Copy build system** - for the docs. We need the Makefile and other scripts that Zephyr uses to generate its docs. We will include these into our repository.
- **Create a basic structure** - so that we can add more docs as the project evolves.
- **Generate HTML and PDF versions** - so that we can have a final doc for delivery and also ability to publish our docs as a website. For private projects it is usually sufficient to have just the PDF.
- **Check that no docs are missing** - this is an important step that many projects miss - you need to enable doxygen errors so that doxygen can detect when you misspell or forget to document parameters of a function and other things.

By the end of this training you will have a working documentation generator that will include existing doxygen comments from our example files into the docs.

Overview

Here is a short list of what we need:

- Doxygen main page template
- PDF header with logo
- Chapter structure
- Inclusion sections for drivers, boards and samples

We will start, as always, by defining what we need as a robot framework script which we can then run and complete the checks for each item in order until we are fully done.

For integrating documentation, we need the following:

```
*** Settings ***
Library  OperatingSystem

*** Variables ***
${ROOT_DIR}  ${CURDIR}/../../

*** Test Cases ***

Documentation generation has been setup
    Documentation infrastructure is in place
    Documentation PDF file is available

Documentation is in good order
    Doxygen main page has been written
    PDF header has been created
    Guides have well structured chapters
    Drivers have been documented
    Libraries have been documented
    Boards have been documented

*** Keywords ***

Documentation infrastructure is in place
    Directory Should Exist  ${ROOT_DIR}/doc/_doxygen/
    Directory Should Exist  ${ROOT_DIR}/doc/_extensions/
    Directory Should Exist  ${ROOT_DIR}/doc/_scripts/
    Directory Should Exist  ${ROOT_DIR}/doc/_static/
    Directory Should Exist  ${ROOT_DIR}/doc/_templates/
    File Should Exist  ${ROOT_DIR}/doc/Makefile
    File Should Exist  ${ROOT_DIR}/doc/CMakeLists.txt
    File Should Exist  ${ROOT_DIR}/doc/conf.py
    File Should Exist  ${ROOT_DIR}/doc/zephyr.doxyfile.in
    File Should Exist  ${ROOT_DIR}/doc/index.rst
    File Should Exist  ${ROOT_DIR}/doc/index-tex.rst

Documentation PDF file is available
    File Should Exist  ${ROOT_DIR}/doc/_build/latex/zephyr.pdf

# we will implement the rest of the keywords in this training
```

Board documentation

- Describe hardware of a board
- Show how to flash and debug the board
- Provide details on supported peripherals

This section represents board documentation and has a slightly different structure from the rest of the docs.

NOTE

Zephyr places board documentation within the board directory itself. This is a workable solution, but I prefer placing documentation inside the docs folder and then using Robot scripts to verify that all docs have been added.

We define the top level keyword that looks first in our boards folder for boards and then checks that we have corresponding docs:

```
Boards have been documented
    @{ARCHS} =      List Directories In Directory   ${ROOT_DIR}/boards/
    FOR  ${ARCH}  IN  @{ARCHS}
        @{BOARDS} =     List Directories In Directory   ${ROOT_DIR}/boards/${ARCH}/
        FOR  ${BOARD}  IN  @{BOARDS}
            File Should Exist  ${ROOT_DIR}/doc/boards/${ARCH}/${BOARD}.rst
            Set Test Variable  ${CHAPTER_FILES}
${ROOT_DIR}/doc/boards/${ARCH}/${BOARD}/
            Chapter follows board reference structure
        END
    END
```

We then add a check to check chapter structure:

```
Chapter follows board reference structure
    Chapter has an overview
    Chapter has a specification of supported hardware
    Chapter covers programming and debugging
    Chapter covers testing
    Chapter lists hardware references

Chapter has an overview
    File Should Exist  ${CHAPTER_FILES}/overview.rst

Chapter has a specification of supported hardware
    File Should Exist  ${CHAPTER_FILES}/hardware.rst

Chapter covers programming and debugging
    File Should Exist  ${CHAPTER_FILES}/programming.rst

Chapter covers testing
    File Should Exist  ${CHAPTER_FILES}/testing.rst

Chapter lists hardware references
    File Should Exist  ${CHAPTER_FILES}/references.rst
```

Overview

This section provides a short explanation of the board and also should mention the following

(if relevant):

- **Photo of the board**
- **Link to data sheet / schematic** - if available

Hardware

This section should describe the hardware of the board.

- **Description of hardware peripherals** - use a list.
- **Pin assignment** - connector pins.
- **Supported external peripherals** - any displays, external peripherals etc. supported by this board. Include wiring diagrams.

Programming and debugging

Provide instructions on how to program a simple application to the board and how to debug it.

- **Command to flash a sample** - using west tool.
- **Connection of debugger** - such as jtag probe.
- **Command to debug** - if "west debug" works then fine - otherwise specify more details.

Links and references

This section includes a set of links that the user can visit to learn more about using this board.

Documentation build system

- Copy zephyr doc build system
- Make adjustments so it runs outside of Zephyr

The first thing we are going to do is copy over zephyr's documentation generation files to our local repository. The benefit of this is that we don't actually want to include the full zephyr documentation into our own project. Instead, we keep zephyr documentation separate - but reuse the tools.

You will need to copy at the very least the following files and directories:

Copy files

```
mkdir doc
cp -r ../zephyr/doc/_doxygen/ doc
cp -r ../zephyr/doc/_extensions/ doc
cp -r ../zephyr/doc/_scripts/ doc
cp -r ../zephyr/doc/_static/ doc
cp -r ../zephyr/doc/_templates/ doc
cp ../zephyr/doc/Makefile doc
cp ../zephyr/doc/CMakeLists.txt doc
cp ../zephyr/doc/conf.py doc
cp ../zephyr/doc/zephyr.doxyfile.in doc
cp ../zephyr/doc/index.rst doc
cp ../zephyr/doc/index-tex.rst doc
```

Make adjustments

- Add project base directory (different from Zephyr base)
- Make sure we use assets from project directory
- Update doxygen template so it doesn't use Zephyr base

Next you need to make modifications to these files to make them compile within your own project.

The full list of changes is below.

The build command that we need to get working is:

```
make -C doc pdf
```

This is how we are going to build our documentation as a PDF file (HTML output is supported as well).

The first thing we need to do is introduce a "PROJECT_BASE" directory alongside of ZEPHYR_BASE. By default the docs generator uses ZEPHYR_BASE to reference files. We don't want this in most cases - but we don't want to delete ZEPHYR_BASE either since we will need Kconfig files from ZEPHYR_BASE alongside of our own files (and other things as well).

doc/CMakeLists.txt

```
diff -u ../zephyr/doc/CMakeLists.txt doc/CMakeLists.txt
--- ../zephyr/doc/CMakeLists.txt    2022-06-16 14:18:11.559642243 +0200
+++ doc/CMakeLists.txt  2022-06-17 14:59:18.304722759 +0200
@@ -5,9 +5,11 @@
```

```
 set(NO_BOILERPLATE TRUE)
 find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE} ..)
+set(PROJECT_BASE "${CMAKE_CURRENT_LIST_DIR}/../")

 file(TO_CMAKE_PATH "${ZEPHYR_BASE}" ZEPHYR_BASE)
 message(STATUS "Zephyr base: ${ZEPHYR_BASE}")
+message(STATUS "Project base: ${PROJECT_BASE}")

 #-------------------------------------------------------------------------------
 # Options
@@ -15,7 +17,7 @@
 set(SPHINXOPTS "-j auto" CACHE STRING "Default Sphinx Options")
 set(LATEXMKOPTS "-halt-on-error -no-shell-escape" CACHE STRING "Default latexmk
options")
 set(DOC_TAG "development" CACHE STRING "Documentation tag")
-set(DTS_ROOTS "${ZEPHYR_BASE}" CACHE STRING "DT bindings root folders")
+set(DTS_ROOTS "${PROJECT_BASE}" CACHE STRING "DT bindings root folders")

 separate_arguments(SPHINXOPTS)
 separate_arguments(LATEXMKOPTS)
@@ -180,8 +182,9 @@
   COMMAND ${CMAKE_COMMAND} -E env
   PYTHONPATH=${ZEPHYR_BASE}/scripts/dts/python-devicetree/src${SEP}$ENV{PYTHONPATH}
   ZEPHYR_BASE=${ZEPHYR_BASE}
+  PROJECT_BASE=${PROJECT_BASE}
   ${PYTHON_EXECUTABLE} ${GEN_DEVICETREE_REST_SCRIPT}
-    --vendor-prefixes ${ZEPHYR_BASE}/dts/bindings/vendor-prefixes.txt
+    --vendor-prefixes ${PROJECT_BASE}/dts/bindings/vendor-prefixes.txt
     ${DTS_ROOT_ARGS}
     ${DOCS_SRC_DIR}/reference/devicetree
   VERBATIM
```

Next we need to make changes to the sphinx config. This file is used to configure the sphinx documentation builder:

```
diff -u ../zephyr/doc/conf.py doc/conf.py
--- ../zephyr/doc/conf.py   2022-06-16 14:18:11.563642243 +0200
+++ doc/conf.py 2022-06-17 23:13:55.861770249 +0200
@@ -11,16 +11,17 @@


 args = get_parser().parse_args()
-ZEPHYR_BASE = Path(__file__).resolve().parents[1]
+ZEPHYR_BASE = Path(str(Path(__file__).resolve().parents[1]) +
"/../zephyr/").resolve()
+PROJECT_BASE = Path(__file__).resolve().parents[1]
 ZEPHYR_BUILD = Path(args.outputdir).resolve()

 # Add the '_extensions' directory to sys.path, to enable finding Sphinx
 # extensions within.
-sys.path.insert(0, str(ZEPHYR_BASE / "doc" / "_extensions"))
+sys.path.insert(0, str(PROJECT_BASE / "doc" / "_extensions"))

 # Add the '_scripts' directory to sys.path, to enable finding utility
 # modules.
-sys.path.insert(0, str(ZEPHYR_BASE / "doc" / "_scripts"))
```

```
+sys.path.insert(0, str(PROJECT_BASE / "doc" / "_scripts"))

 # Add the directory which contains the runners package as well,
 # for autodoc directives on runners.xyz.
@@ -35,19 +36,17 @@

 # -- Project -------------------------------------------------------------

-project = "Zephyr Project"
-copyright = "2015-2022 Zephyr Project members and individual contributors"
-author = "The Zephyr Project Contributors"
+project = "Example Project Documentation"
+copyright = "2015-2022 Example Project members and individual contributors"
+author = "The Example Project Contributors"

 # parse version from 'VERSION' file
-with open(ZEPHYR_BASE / "VERSION") as f:
+with open(PROJECT_BASE / "VERSION") as f:
     m = re.match(
         (
             r"^VERSION_MAJOR\s*=\s*(\d+)$\n"
             + r"^VERSION_MINOR\s*=\s*(\d+)$\n"
-            + r"^PATCHLEVEL\s*=\s*(\d+)$\n"
-            + r"^VERSION_TWEAK\s*=\s*\d+$\n"
-            + r"^EXTRAVERSION\s*=\s*(.*)$"
+            + r"^VERSION_PATCHLEVEL\s*=\s*(\d+)$\n"
         ),
         f.read(),
         re.MULTILINE,
@@ -57,10 +56,8 @@
         sys.stderr.write("Warning: Could not extract kernel version\n")
         version = "Unknown"
     else:
-        major, minor, patch, extra = m.groups(1)
+        major, minor, patch = m.groups(1)
         version = ".".join((major, minor, patch))
-        if extra:
-            version += "-" + extra

 release = version

@@ -86,8 +83,8 @@
 ]

 # Only use SVG converter when it is really needed, e.g. LaTeX.
-if tags.has("svgconvert"):  # pylint: disable=undefined-variable
-    extensions.append("sphinxcontrib.rsvgconverter")
+#if tags.has("svgconvert"):  # pylint: disable=undefined-variable
+#    extensions.append("sphinxcontrib.rsvgconverter")

 templates_path = ["_templates"]

@@ -105,7 +102,6 @@
 numfig = True

 rst_epilog = """
```

```
-.. include:: /substitutions.txt
 """

 # -- Options for HTML output -------------------------------------------------
@@ -117,15 +113,15 @@
     "prev_next_buttons_location": None
 }
 html_title = "Zephyr Project Documentation"
-html_logo = str(ZEPHYR_BASE / "doc" / "_static" / "images" / "logo.svg")
-html_favicon = str(ZEPHYR_BASE / "doc" / "_static" / "images" / "favicon.png")
-html_static_path = [str(ZEPHYR_BASE / "doc" / "_static")]
+html_logo = str(PROJECT_BASE / "doc" / "_static" / "images" / "logo.svg")
+html_favicon = str(PROJECT_BASE / "doc" / "_static" / "images" / "favicon.png")
+html_static_path = [str(PROJECT_BASE / "doc" / "_static")]
 html_last_updated_fmt = "%b %d, %Y"
 html_domain_indices = False
 html_split_index = True
 html_show_sourcelink = False
 html_show_sphinx = False
-html_search_scorer = str(ZEPHYR_BASE / "doc" / "_static" / "js" / "scorer.js")
+html_search_scorer = str(PROJECT_BASE / "doc" / "_static" / "js" / "scorer.js")

 is_release = tags.has("release")  # pylint: disable=undefined-variable
 reference_prefix = ""
@@ -139,13 +135,6 @@
     "current_version": version,
     "versions": (
         ("latest", "/"),
-        ("3.0.0", "/3.0.0/"),
-        ("2.7.0", "/2.7.0/"),
-        ("2.6.0", "/2.6.0/"),
-        ("2.5.0", "/2.5.0/"),
-        ("2.4.0", "/2.4.0/"),
-        ("2.3.0", "/2.3.0/"),
-        ("1.14.1", "/1.14.1/"),
     ),
     "display_vcs_link": True,
     "reference_links": {
@@ -159,8 +148,8 @@

 latex_elements = {
     "papersize": "a4paper",
-    "maketitle": open(ZEPHYR_BASE / "doc" / "_static" / "latex" /
"title.tex").read(),
-    "preamble": open(ZEPHYR_BASE / "doc" / "_static" / "latex" /
"preamble.tex").read(),
+    "maketitle": open(PROJECT_BASE / "doc" / "_static" / "latex" /
"title.tex").read(),
+    "preamble": open(PROJECT_BASE / "doc" / "_static" / "latex" /
"preamble.tex").read(),
     "fontpkg": r"\usepackage{charter}",
     "sphinxsetup": ",".join(
         (
@@ -174,18 +163,22 @@
         )
```

```
     ),
 }
-latex_logo = str(ZEPHYR_BASE / "doc" / "_static" / "images" / "logo-latex.pdf")
+latex_logo = str(PROJECT_BASE / "doc" / "_static" / "images" / "logo-latex.pdf")
 latex_documents = [
-    ("index-tex", "zephyr.tex", "Zephyr Project Documentation", author, "manual"),
+    ("index-tex", "zephyr.tex", "Example Project Documentation", author, "manual"),
 ]

 # -- Options for zephyr.doxyrunner plugin ---------------------------------

 doxyrunner_doxygen = os.environ.get("DOXYGEN_EXECUTABLE", "doxygen")
-doxyrunner_doxyfile = ZEPHYR_BASE / "doc" / "zephyr.doxyfile.in"
+doxyrunner_doxyfile = PROJECT_BASE / "doc" / "zephyr.doxyfile.in"
 doxyrunner_outdir = ZEPHYR_BUILD / "doxygen"
 doxyrunner_fmt = True
-doxyrunner_fmt_vars = {"ZEPHYR_BASE": str(ZEPHYR_BASE), "ZEPHYR_VERSION": version}
+doxyrunner_fmt_vars = {
+   "ZEPHYR_BASE": str(ZEPHYR_BASE),
+   "PROJECT_BASE": str(PROJECT_BASE),
+   "PROJECT_VERSION": version
+}
 doxyrunner_outdir_var = "DOXY_OUT"

 # -- Options for Breathe plugin ------------------------------------------
@@ -245,17 +238,17 @@
 # -- Options for zephyr.external_content ----------------------------------

 external_content_contents = [
-    (ZEPHYR_BASE / "doc", "[!_]*"),
-    (ZEPHYR_BASE, "boards/**/*.rst"),
-    (ZEPHYR_BASE, "boards/**/doc"),
-    (ZEPHYR_BASE, "samples/**/*.rst"),
-    (ZEPHYR_BASE, "samples/**/doc"),
+    (PROJECT_BASE / "doc", "[!_]*"),
+    (PROJECT_BASE, "boards/**/*.rst"),
+    (PROJECT_BASE, "boards/**/doc"),
+    (PROJECT_BASE, "samples/**/*.rst"),
+    (PROJECT_BASE, "samples/**/doc"),
 ]
 external_content_keep = [
-    "reference/kconfig/*",
-    "reference/devicetree/bindings.rst",
-    "reference/devicetree/bindings/**/*",
-    "reference/devicetree/compatibles/**/*",
+    #"reference/kconfig/*",
+    #"reference/devicetree/bindings.rst",
+    #"reference/devicetree/bindings/**/*",
+    #"reference/devicetree/compatibles/**/*",
 ]
```

Inside the "index-tex.rst" and "index.rst" we remove the zephyr text and add our own toctable list of files we want to include. These two files are used for PDF and HTML docs.

```
Example Project Documentation
#############################
```

```
.. toctree::
   :maxdepth: 1
   :caption: Contents

   guides/index.rst
   boards/index.rst
   drivers/index.rst
   lib/index.rst
```

Lastly we modify the doxygen generator file. Most important point here is that we enable treating all warnings as errors and we also want to enable errors for undocumented symbols. It's really important that we don't ignore this because undocumented symbols result in substandard docs.

```
diff -u ../zephyr/doc/zephyr.doxyfile.in doc/zephyr.doxyfile.in
--- ../zephyr/doc/zephyr.doxyfile.in    2022-06-16 14:18:11.635642243 +0200
+++ doc/zephyr.doxyfile.in  2022-06-17 14:59:18.304722759 +0200
@@ -32,26 +32,26 @@
 # title of most generated pages and in a few other places.
 # The default value is: My Project.

-PROJECT_NAME           = "Zephyr API Documentation"
+PROJECT_NAME           = "Example Project"

 # The PROJECT_NUMBER tag can be used to enter a project or revision number. This
 # could be handy for archiving the generated documentation or if some version
 # control system is used.

-PROJECT_NUMBER         = @ZEPHYR_VERSION@
+PROJECT_NUMBER         = @PROJECT_VERSION@

 # Using the PROJECT_BRIEF tag one can provide an optional one line description
 # for a project that appears at the top of each page and should give viewer a
 # quick idea about the purpose of the project. Keep the description short.

-PROJECT_BRIEF          = "A Scalable Open Source RTOS"
+PROJECT_BRIEF          = "Example project brief"

 # With the PROJECT_LOGO tag one can specify a logo or an icon that is included
 # in the documentation. The maximum height of the logo should not exceed 55
 # pixels and the maximum width should not exceed 200 pixels. Doxygen will copy
 # the logo to the output directory.

-PROJECT_LOGO           = @ZEPHYR_BASE@/doc/_doxygen/logo.svg
+PROJECT_LOGO           = @PROJECT_BASE@/doc/_doxygen/logo.svg

 # The OUTPUT_DIRECTORY tag is used to specify the (relative or absolute) path
 # into which the generated documentation will be written. If a relative path is
@@ -481,7 +481,7 @@
 # normally produced when WARNINGS is set to YES.
 # The default value is: NO.

-EXTRACT_ALL            = YES
+EXTRACT_ALL            = NO
```

```
 # If the EXTRACT_PRIVATE tag is set to YES, all private members of a class will
 # be included in the documentation.
@@ -838,7 +838,7 @@
 # WARN_IF_INCOMPLETE_DOC
 # The default value is: NO.

-WARN_NO_PARAMDOC       = NO
+WARN_NO_PARAMDOC       = YES

 # If the WARN_AS_ERROR tag is set to YES then doxygen will immediately stop when
 # a warning is encountered. If the WARN_AS_ERROR tag is set to FAIL_ON_WARNINGS
@@ -847,7 +847,7 @@
 # Possible values are: NO, YES and FAIL_ON_WARNINGS.
 # The default value is: NO.

-WARN_AS_ERROR          = NO
+WARN_AS_ERROR          = YES

 # The WARN_FORMAT tag determines the format of the warning messages that doxygen
 # can produce. The string should contain the $file, $line, and $text tags, which
@@ -875,13 +875,11 @@
 # spaces. See also FILE_PATTERNS and EXTENSION_MAPPING
 # Note: If this tag is empty the current directory is searched.

-INPUT                  = @ZEPHYR_BASE@/doc/_doxygen/mainpage.md \
-                         @ZEPHYR_BASE@/doc/_doxygen/groups.dox \
-                         @ZEPHYR_BASE@/kernel/include/kernel_arch_interface.h \
-                         @ZEPHYR_BASE@/include/ \
-                         @ZEPHYR_BASE@/lib/libc/minimal/include/ \
-                         @ZEPHYR_BASE@/subsys/testsuite/ztest/include/ \
-                         @ZEPHYR_BASE@/tests/kernel/
+INPUT                  = @PROJECT_BASE@/doc/_doxygen/mainpage.md \
+                         @PROJECT_BASE@/doc/_doxygen/groups.dox \
+                         @PROJECT_BASE@/include/ \
+                         @PROJECT_BASE@/drivers/ \
+                         @PROJECT_BASE@/lib/ \

 # This tag can be used to specify the character encoding of the source files
 # that doxygen parses. Internally doxygen uses the UTF-8 encoding. Doxygen uses
@@ -1230,7 +1228,7 @@
 # of the possible markers and block names see the documentation.
 # This tag requires that the tag GENERATE_HTML is set to YES.

-HTML_HEADER            = @ZEPHYR_BASE@/doc/_doxygen/header.html
+HTML_HEADER            = @PROJECT_BASE@/doc/_doxygen/header.html

 # The HTML_FOOTER tag can be used to specify a user-defined HTML footer for each
 # generated HTML page. If the tag is left blank doxygen will generate a standard
@@ -1240,7 +1238,7 @@
 # that doxygen normally uses.
 # This tag requires that the tag GENERATE_HTML is set to YES.

-HTML_FOOTER            = @ZEPHYR_BASE@/doc/_doxygen/footer.html
+HTML_FOOTER            = @PROJECT_BASE@/doc/_doxygen/footer.html

 # The HTML_STYLESHEET tag can be used to specify a user-defined cascading style
```

```
 # sheet that is used by each HTML page. It can be used to fine-tune the look of
@@ -1265,10 +1263,10 @@
 # list). For an example see the documentation.
 # This tag requires that the tag GENERATE_HTML is set to YES.

-HTML_EXTRA_STYLESHEET  = @ZEPHYR_BASE@/doc/_doxygen/doxygen-awesome.css \
-                         @ZEPHYR_BASE@/doc/_doxygen/doxygen-awesome-sidebar-only.css
\
-                         @ZEPHYR_BASE@/doc/_doxygen/doxygen-awesome-sidebar-only-
darkmode-toggle.css \
-                         @ZEPHYR_BASE@/doc/_doxygen/doxygen-awesome-zephyr.css
+HTML_EXTRA_STYLESHEET  = @PROJECT_BASE@/doc/_doxygen/doxygen-awesome.css \
+                         @PROJECT_BASE@/doc/_doxygen/doxygen-awesome-sidebar-
only.css \
+                         @PROJECT_BASE@/doc/_doxygen/doxygen-awesome-sidebar-only-
darkmode-toggle.css \
+                         @PROJECT_BASE@/doc/_doxygen/doxygen-awesome-zephyr.css

 # The HTML_EXTRA_FILES tag can be used to specify one or more extra images or
 # other source files which should be copied to the HTML output directory. Note
@@ -1278,8 +1276,8 @@
 # files will be copied as-is; there are no commands or markers available.
 # This tag requires that the tag GENERATE_HTML is set to YES.

-HTML_EXTRA_FILES       = @ZEPHYR_BASE@/doc/_doxygen/doxygen-awesome-darkmode-
toggle.js \
-                         @ZEPHYR_BASE@/doc/_doxygen/doxygen-awesome-zephyr.js
+HTML_EXTRA_FILES       = @PROJECT_BASE@/doc/_doxygen/doxygen-awesome-darkmode-
toggle.js \
+                         @PROJECT_BASE@/doc/_doxygen/doxygen-awesome-zephyr.js

 # The HTML_COLORSTYLE_HUE tag controls the color of the HTML output. Doxygen
 # will adjust the colors in the style sheet and background images according to
```

Documentation chapter types

Documentation consists of several types of documents:

- **Guides** - these are long form articles which achieve some specific result. A guide becomes a large chapter inside the final PDF book. Guides consist of sections (numbered 1.1 etc.) and subsections (numbered 1.1.1 etc.)
- **Board docs** - these describe board support and have a specific set of sections that we want to capture in our structure verification scripts.
- **References** - these are sections that follow the pattern of a data sheet. These discuss code of a particular subsystem, providing implementation details. References are divided into subsystems. Each subsystem gets its own chapter in the PDF book (top level section). Subsystems consist of specific APIs that are grouped into sections.
- **Release notes** - these are comments that we add to a release at the end. These need to be cross referenced against git commits.

By establishing a pattern for each type of section, we can verify that we haven't missed to add important parts to each subsection of our documentation.

It helps to identify the possible types of sections we may have based on zephyr directory structure:

- **modules/<module>**
- **soc/<arch>/<family>/<model>**
- **drivers/<class>/<device>**
- **subsys/<name>**
- **lib/<name>**
- **boards/<arch>/<board>/**
- **samples/<category>/<name>**
- **arch/<name>**
- **tests/<path>**

We will focus on only drivers, libraries and boards as part of this guide.

- **drivers/<class>/<device>**
- **lib/<name>**
- **boards/<arch>/<board>/**

Guide chapters

- Follow tutorial style
- Describe a story of getting a particular outcome

We will start with our guide chapters:

```
doc/guides/ ❶
└─── <chapter - guide name> ❷
    ├─── index.rst ❸
    ├─── <section> ❹
    │    ├─── implementation.rst
    │    ├─── offer.rst
    │    ├─── pitfalls.rst
    │    ├─── prerequisites.rst
    │    ├─── what-is-this.rst
    │    ├─── what-we-have-done.rst
    │    └─── what-we-will-do.rst
    └─── introduction.rst ❺
```

❶ - we place all guides under doc/guides folder
❷ - the guides folder contains chapters - one per guide
❸ - each chapter contains an index file that includes all other files
❹ - each section of the chapter has a folder with partials
❺ - each section has its own file that has the same name as the folder

We start with a directory for the guide chapter, which then contains a directory per section. Inside each section directory we have a set of files that represent parts of the section that have to be included. In this way we can apply the same pattern to all chapters and make sure that we do not forget anything.

```
Guides have well structured chapters
    @{GUIDES} =     List Directories In Directory   ${ROOT_DIR}/doc/guides/
    FOR  ${GUIDE}  IN  @{GUIDES}
        File Should Exist  ${ROOT_DIR}/doc/guides/${GUIDE}.rst
        Set Test Variable  ${CHAPTER_FILES}  ${ROOT_DIR}/doc/guides/${GUIDE}/
        Chapter follows guide structure
    END
```

To check the guide structure we define the keyword:

```
Chapter follows guide structure
    Chapter has a small introduction
    Chapter covers prerequisite knowledge
    Chapter covers goal of this chapter
    Chapter covers action steps for achieving the goal
    Chapter covers pitfalls along the way
    Chapter covers what has been achieved
    Chapter has an offer at the end

Chapter covers prerequisite knowledge
    File Should Exist  ${CHAPTER_FILES}/prerequisites.rst

Chapter covers goal of this chapter
    File Should Exist  ${CHAPTER_FILES}/what-we-will-do.rst

Chapter covers action steps for achieving the goal
```

```
    File Should Exist  ${CHAPTER_FILES}/implementation.rst

Chapter covers pitfalls along the way
    File Should Exist  ${CHAPTER_FILES}/pitfalls.rst

Chapter covers what has been achieved
    File Should Exist  ${CHAPTER_FILES}/what-we-have-done.rst

Chapter has an offer at the end
    File Should Exist  ${CHAPTER_FILES}/offer.rst
```

Prerequisites

This section should tell the reader of the guide what prior knowledge they need to have in order to have full understanding of the new information being presented. This can be a list of links to other guides or references.

What we will do

This section should make the reader envision a single particular result that the guide is going to achieve. This is a "promise" to the reader about what will come.

Action steps

This is the bulk of the content of the guide that all leads to achieving the promise stated earlier.

Pitfalls

This is an important section that outlines possible pitfalls that the user may run into when trying to implement the guide on their own.

What has been achieved

This section reiterates over what has been accomplished and possibly also shows some kind of proof (like screenshots) of the results of the steps if done correctly.

Offer

The user reads your guide because they have a problem they need solved. The least you can do is offer to solve the problem for them. This section is a good place to make this offer and explain to the reader what they need to do next in order to get it.

Reference chapters

- Describe API reference
- Based on code structure

This chapter template is used for drivers and libraries.

For drivers the checker looks a little more complex because it needs to take into account driver classes:

```
Drivers have been documented
    @{DRIVER_CLASSES} =     List Directories In Directory   ${ROOT_DIR}/drivers/
    FOR  ${DRIVER_CLASS}  IN  @{DRIVER_CLASSES}
        Directory Should Exist  ${ROOT_DIR}/doc/drivers/${DRIVER_CLASS}/
        @{DRIVERS} =    List Files In Directory  ${ROOT_DIR}/drivers/${DRIVER_CLASS}/
*.c
        FOR  ${DRIVER}  IN  @{DRIVERS}
            ${PATH}  ${FILE}  Split Path  ${DRIVER}
            ${SECTION}  ${EXT}  Split Extension  ${FILE}
            Set Test Variable  ${CHAPTER_FILES}
${ROOT_DIR}/doc/drivers/${DRIVER_CLASS}/${SECTION}/
            File Should Exist  ${ROOT_DIR}/doc/drivers/${DRIVER_CLASS}/${SECTION}.rst
            Chapter follows reference structure
        END
        # find invalid directories
        @{INVALIDS} =   List Directories In Directory
${ROOT_DIR}/drivers/${DRIVER_CLASS}/
        FOR  ${INVALID}  IN  @{INVALIDS}
            Log To Console  Do not put directories under driver class folder
(drivers/${DRIVER_CLASS}/${INVALID})
            Fail
        END
    END

Libraries have been documented
    @{LIBRARIES} =  List Directories In Directory   ${ROOT_DIR}/lib/
    FOR  ${LIB}  IN  @{LIBRARIES}
        File Should Exist  ${ROOT_DIR}/doc/lib/${LIB}.rst
        Set Test Variable  ${CHAPTER_FILES}  ${ROOT_DIR}/doc/lib/${LIB}/
        Chapter follows reference structure
    END
```

The reference chapter check looks as follows:

```
Chapter follows reference structure
    Chapter has a small introduction
    Chapter has summary of main features
    Chapter has functional description
    Chapter has usage examples
    Chapter has doxygen reference

Chapter has a small introduction
    File Should Exist  ${CHAPTER_FILES}/what-is-this.rst

Chapter has summary of main features
    File Should Exist  ${CHAPTER_FILES}/main-features.rst
```

```
Chapter has functional description
    File Should Exist  ${CHAPTER_FILES}/description.rst

Chapter has usage examples
    File Should Exist  ${CHAPTER_FILES}/usage.rst

Chapter has doxygen reference
    File Should Exist  ${CHAPTER_FILES}/api-reference.rst
```

What this is

This section should contain a short introduction of what this is about.

Main features

Here you describe the main features of the driver or the library component.

Description

This is a more detailed description of all features.

Usage examples

This section should contain case studies of using the driver to achieve desired results. This section should focus on samples and use case scenarios.

API reference

This should directly include the doxygen group of the driver that contains driver api reference. To do this you need to place all your public methods and structures into a doxygen group and then reference this group in your docs:

```
/*!
 * @defgroup example-driver-api Example Driver API
 * @{
 **/
/// code goes here
/*!
 * @}
 **/
```

Then in your rst file you do this:

```
API Reference
*************

.. doxygengroup:: example-driver-api
```

Linking everything into one structure

- We use toc tree directive
- Files are included from doc "build" tree
- External files are automatically copied to build tree

When you are organizing your docs in this hierarchical way you will need to add intermediate index.rst files to folders that in turn include other rst files from subdirectories.

You should use the toctree directive for this:

```
Drivers
#######

.. toctree::
   :maxdepth: 1
   :caption: Drivers

   example/example_driver.rst
```

Pitfalls

Documentation generation is a complex process with many moving parts. Zephyr has even made it use a separate Makefile outside of the main build process. The main pitfalls you are likely to run into are the following:

- **Forgetting to adjust every detail in scripts** - basically moving zephyr files to your project involves making many small changes to them such as setting your project name, changing ZEPHYR_BASE to your own project path etc. To see all changes do a diff meld between this getting started guide source code repository and the zephyr docs version: "meld doc ../zephyr/doc"
- **Forgetting to use toctree** - when you are including files from sub-folders you can not use the include directive - you have to use toctree to ensure they are properly added to the hierarchy of sections.
- **Not adding doc build to the CI script** - the doc build needs to be part of the CI. While doing this chapter I discovered that I forgot to add the build process to CI in the last module. I've added it in the MR for this chapter!

Success metrics

- Beautifully looking documentation
- Easy to include references to code
- External documentation can be included in the build as well
- Documentation is buildable in multiple formats (at least HTML and PDF)

The biggest success metric for implementing these steps correctly is that you are able to generate both the PDF and HTML docs for the whole project. The docs contain all sections and the documentation tree has a well defined structure.

Summary

In this module we have covered adding documentation generation to your firmware project.

We have also defined:

- **Guide section structure** - verified by the robot framework scripts.
- **Driver and library reference structure**
- **Board documentation reference structure**
- **Library documentation structure**

Documentation is now also generated for all source code.

Getting Help

- Attend live calls: https://swedishembedded.com/live
- Get consulting: https://swedishembedded.com/go

If you think that this was useful and can bring value to your project, but do not want to implement this yourself then Swedish Embedded Group would be happy to help you get this done.

It can be a daunting task to migrate legacy firmware to a well oiled system like Zephyr. By working with us you get:

- **Quality but quick solutions to your firmware challenges** - so that you can solve your firmware challenges quickly.
- **Training of your team** - so that your team can work faster on development of firmware supports for your products.
- **Reuse existing code to the maximum** - we believe in keeping things open so that we can work collaboratively using open source technology.

If you think this sounds good, click here to book a discovery call.

```
Author: Martin Schröder, 17 Jun 2022
Email: martin.schroder@swedishembedded.com
Training: Zephyr Getting Started Training
Module: 4
```

Chapter 5

Test Infrastructure

► https://www.youtube.com/watch?v=qbkZjOMOt_w *(YouTube video)*

Recap

- We have a repository
- We have CI setup and working with docker image
- We have an organized directory structure
- We are able to generate docs efficiently

At this point our repository is in quite a good shape. However at least one very important part is missing.

Problems we still have

- No test infrastructure in place
- No automatic way to verify code coverage

We still don't have testing in place. In particular, we don't have an automated way to build and run tests and to generate and verify code coverage afterwards.

Also several important questions are still unanswered:

- Should we check code coverage for all tests?
- What is a good threshold?

We still need to answer these and we still need to add testing infrastructure.

Additionally, we are quite limited in Zephyr with the kinds of tests we can run - even with existing ZTest and FFF support.

Zephyr mainly supports cross platform integration testing and has only limited support for unit tests. Zephyr has a few more problems that are currently not solved:

- **Zephyr has no way to guarantee logical integrity** - because there is no hard requirement for 100% branch and line coverage. We need this if we are going to be able to say that our code is "logically fully correct" (coverage is not a guarantee but at least it is a solid step in the right direction).
- **Zephyr doesn't natively support system testing** - meaning there is no way to run user case scenarios using a system like RobotFramework in Zephyr. We need this if we are going to verify that our software complies with actual use case scenarios.

Zephyr does however have integration testing and this is a very good feature. Zephyr integration testing is mainly done on native_posix platform which means that your code is compiled using local linux compiler and runs on your local system. Zephyr uses simulation drivers and a custom timing implementation for posix that allows us to run tests in "zero time" where your application skips all idle time. This makes your tests execute significantly faster than if you were to run them on actual hardware (a test that sleeps 60 seconds would run instantly but would behave "as though" it slept for 60 seconds).

Solution proposal

- Use swedish embedded CMock support provided by: https://github.com/swedishembedded/testing
- Add testing to CI with coverage checking

In this module we are going to implement unit, integration and system testing infrastructure for our embedded project.

- **Unit tests** - these will verify logical integrity of our software.
- **Integration tests** - these will verify structural integrity of our software.
- **System tests** - these will verify our software against user requirements.

All three of these together will ensure that every task is in a fully deliverable state before it is merged into our main branch.

To solve testing we must:

- **Adding CMock as a mocking framework** - this will allow us to do unit testing of ANY code - not just the simplest modules.
- **Changing from ZTest to Unity** - since CMock is already using Unity assertion macros to verify that tests pass, you should use Unity to keep your tests consistent.
- **Renode pipeline** - we will also extend our CI pipeline to run user requirements written in Robot Framework against our generated executables.

Let's get right into it and map out what we are going to accomplish in this module!

Mapping out the tasks

A good starting point will be this:

```
Testing infrastructure has been fully setup
    Check unity enabled
    Check cmock enabled
    There is a Kconfig option to enable unit testing
    There is a unity config
    There is a zephyr specific tear down that works with twister
    There is support for unity command line arguments in tests
    There is a code coverage check
```

We can define some main tasks that we need to accomplish today:

- **Add unity and CMock as sub-modules to Zephyr** - so that we can use these libraries directly after "west update" in our project.
- **Add CMake and Kconfig options** - these will define settings for our unity and CMock modules.
- **Add glue code** - we need this in order to support command line arguments and be able to detect failed tests when we are running tests through Twister.
- **Add tests for existing code in the repository** - we must add these before we can merge this MR so that all checks can be enabled.
- **Enable coverage threshold check** - we will enable coverage threshold so that we can enforce that all code is tested before it can be merged.

While this has been greatly simplified with the introduction of swedish embedded testing toolkit, we are still for now going to go over the implementation because it gives a deeper understanding of what is involved.

Defining directory structure

- Create a module for CMock and Unity
- Create unity configuration
- Create C teardown code

First we need to decide where we put our unity glue code and scripts configuration files. This is constrained by the fact that we don’t want to "unit test" this code. Neither do we want to enforce coverage checks on it.

So we can not place it under any directory that we want to enforce these checks on. This means that we definitely can not place this code under "drivers" or "lib".

If you look at the Zephyr tree, you will find that zephyr has a "modules" directory. This directory contains Kconfig files for modules which are in the upper level "../modules" directory in the workspace.

Since both CMock and Unity are external projects, it makes the most sense for us to treat them just like all the other modules.

We will need to make sure that we have at least the following files:

scrum/infrastructure/testing.robot

```
Directory structure has been setup
    CMock is placed in expected location
    Unity is placed in expected location
    Unity config has been created
    Unity twister compliant teardown has been created
    Unity cmake and kconfig have been created

CMock is placed in expected location
    Directory Should Exist  ${ROOT_DIR}/../modules/test/cmock/

Unity is placed in expected location
    Directory Should Exist  ${ROOT_DIR}/../modules/test/cmock/vendor/unity/

Unity config has been created
    File Should Exist  ${ROOT_DIR}/modules/unity/unity_cfg.yml
    File Should Exist  ${ROOT_DIR}/modules/unity/unity_config.h

Unity twister compliant teardown has been created
    File Should Exist  ${ROOT_DIR}/modules/unity/test_teardown.c

Unity cmake and kconfig have been created
    File Should Exist  ${ROOT_DIR}/modules/unity/Kconfig
    File Should Exist  ${ROOT_DIR}/modules/unity/CMakeLists.txt
```

CMock and Unity

- Verify integrity of configuration
- Update west uml

The first thing we need to do is add unity and cmock to our west.yaml file:

To verify that this step has been completed we will add keywords for checking that unity and cmock have been added to west.yaml:

scrum/infrastructure/testing.py

```
#!/usr/bin/env python3
import os
import yaml
from robot.libraries.BuiltIn import BuiltIn

PROJECT_ROOT = os.path.realpath(os.path.dirname(os.path.realpath(__file__)) +
"/../../")

def check_unity_enabled():
    yaml_file = PROJECT_ROOT + "/west.yml"
    with open(yaml_file, "r") as stream:
        yml = yaml.safe_load(stream)
        unity = next((x for x in yml["manifest"]["projects"] if x["name"] ==
"unity"), None)
        if(not unity):
          raise BaseException("You need to add unity project to west.yml")
        if(unity["path"] != "modules/test/cmock/vendor/unity"):
          raise BaseException("Please place unity under %s" % (expected_unity_path))

def check_cmock_enabled():
    yaml_file = PROJECT_ROOT + "/west.yml"
    with open(yaml_file, "r") as stream:
        yml = yaml.safe_load(stream)
        cmock = next((x for x in yml["manifest"]["projects"] if x["name"] ==
"cmock"), None)
        if(not cmock):
          raise BaseException("You need to add cmock project to west.yml")
        if(cmock["path"] != "modules/test/cmock"):
          raise BaseException("Please place cmock under %s" % (expected_cmock_path))
```

Let's now go ahead and add the necessary settings:

```
--- a/west.yml
+++ b/west.yml
@@ -8,9 +8,19 @@ manifest:
   remotes:
     - name: zephyrproject-rtos
       url-base: https://github.com/zephyrproject-rtos
+    - name: throwtheswitch
+      url-base: https://github.com/ThrowTheSwitch

   projects:
     - name: zephyr
       remote: zephyrproject-rtos
       revision: v3.0.0
```

```
      import: true
+   - name: cmock
+     path: modules/test/cmock
+     revision: 1b81269e78a5ecc15fd7befa95fe3a1fc19c41ad
+     remote: throwtheswitch
+   - name: unity
+     path: modules/test/cmock/vendor/unity
+     revision: 1958b977019c3ac19bb83350f7b26e60cbeb6c75
+     remote: throwtheswitch
```

Here we are adding two repositories:

- https://github.com/ThrowTheSwitch/cmock
- https://github.com/ThrowTheSwitch/unity

These get added to our workspace in "../modules/test/" directory relative to our project.

Now we are ready to add unity configuration.

Configuring CMock and Unity

- Easy way: https://github.com/swedishembedded/testing
- Hard way: read training materials

Adding unity support requires us to add a few files to configure unity and also enable us to use unity in our Zephyr tests.

The first file we need to create is the unity configuration. Since unity can be used without CMock, we don't include any CMock options into this file.

modules/unity/unity_cfg.yaml

```
:unity:
    :suite_teardown: >
       extern int test_suite_teardown(int); return test_suite_teardown(num_failures);
    :main_name: unity_main
    :cmdline_args: true
```

Here we want to do three things:

- **Configure teardown function** - this is needed because Twister looks in process STDOUT for string "PROJECT EXECUTION SUCCESSFUL" to see if a test was a success or failure. Therefore we need to add a custom exit function so that we can do this printout.
- **Set a name for unity main** - we will create a wrapper for our main function and call this from our wrapper.
- **Enable command line options** - we will need this in order to make it possible to pass command line options to our test executable that uses unity.

The next file we are going to add is a header that will define compilation options for unity:

modules/unity/unity_config.h

```
#ifndef UNITY_CONFIG_H__
#define UNITY_CONFIG_H__

#define UNITY_OUTPUT_COLOR
#define UNITY_USE_COMMAND_LINE_ARGS

#ifndef CONFIG_BOARD_NATIVE_POSIX
#include <stddef.h>
#include <stdio.h>
#define UNITY_EXCLUDE_SETJMP_H 1
#define UNITY_OUTPUT_CHAR(a) printk("%c", a)
#endif

extern int unity_main(int argc, char **argv);

#endif
```

This file defines how unity will print characters to the output. We also need to enable command line here as well. We will instruct unity to include this file from the CMake file below.

Next we need to add code for main and teardown:

modules/unity/main.c

```
#include <zephyr.h>
#include <unity.h>
```

```
#ifdef CONFIG_BOARD_NATIVE_POSIX
#include "posix_board_if.h"
#include "cmdline.h"
#endif

void main(void)
{
#ifdef CONFIG_BOARD_NATIVE_POSIX
    printk("Parsing command line arguments\n");
    int argc;
    char **argv;

    native_get_test_cmd_line_args(&argc, &argv);
    posix_exit(unity_main(argc, argv));
#else
    printk("Ignoring command line arguments\n");
    unity_main(0, NULL);
#endif
}
```

This defines the entry point and passes command line arguments to unity. There is one issue with this implementation: the zephyr "native_get_test_cmd_line_args" function returns just the command line options - while unity expects first argument to be name of the executable (which does not exist when we are running on plain hardware) - the solution is that you actually pass name of the executable as first argument to "-testargs" option which is implemented by Zephyr (the value doesn't matter - you can pass "dummy" if you want).

The teardown function that makes our test work with twister looks like this:

modules/unity/main.c

```
#include <zephyr.h>
#ifdef CONFIG_BOARD_NATIVE_POSIX
#include "posix_board_if.h"
#endif

int generic_test_suite_teardown(int num_failures)
{
    int ret = num_failures > 0 ? 1 : 0;
    printk("PROJECT EXECUTION %s\n", (num_failures) ? "FAILED" : "SUCCESSFUL");
    #ifdef CONFIG_BOARD_NATIVE_POSIX
    posix_exit(ret);
    #endif
    return ret;
}

__weak int __unused test_suite_teardown(int num_failures)
{
    return generic_test_suite_teardown(num_failures);
}
```

Lastly, we need to add the CMakeLists.txt file that will provide us with functions to generate the test runner:

modules/unity/CMakeLists.txt

```
zephyr_library()
```

```
# locate unity directory
set_property(GLOBAL PROPERTY UNITY_DIR
${ZEPHYR_BASE}/../modules/test/cmock/vendor/unity)
# need to also import it into current file so it can be used
get_property(UNITY_DIR GLOBAL PROPERTY UNITY_DIR)

# locate unity config file
set(UNITY_CONFIG_FILE
        ${CMAKE_CURRENT_LIST_DIR}/unity_cfg.yaml
        CACHE STRING "")

# check if we have ruby executable
find_program(RUBY_PATH ruby)
if(${RUBY_PATH} STREQUAL RUBY_PATH-NOTFOUND)
        message(FATAL_ERROR "Ruby executable not found")
endif()

# add unity include directories to our build
zephyr_include_directories(${UNITY_DIR}/src ${CMAKE_CURRENT_LIST_DIR})

# add unity sources
zephyr_library_sources(${UNITY_DIR}/src/unity.c)

# add glue source code to build
zephyr_library_sources(teardown.c)
zephyr_library_sources(main.c)

# instruct unity to include "unity_config.h" file
zephyr_compile_definitions(UNITY_INCLUDE_CONFIG_H)

# function for generating the test runner
function(unity_generate_test_runner TEST_FILE_PATH)
    # get the global setting
    get_property(UNITY_DIR GLOBAL PROPERTY UNITY_DIR)
    # will be placed in build/runner
    set(UNITY_OUTPUT_DIR ${APPLICATION_BINARY_DIR}/runner)
    # create the directory
    file(MAKE_DIRECTORY "${UNITY_OUTPUT_DIR}")
    # get filename with extension
    get_filename_component(TEST_FILE_NAME "${TEST_FILE_PATH}" NAME)
    set(OUTPUT_FILE_PATH "${UNITY_OUTPUT_DIR}/runner_${TEST_FILE_NAME}")
    # add command for building this runner file
    add_custom_command(
        COMMAND
            ${RUBY_PATH} ${UNITY_DIR}/auto/generate_test_runner.rb
            ${UNITY_CONFIG_FILE} ${TEST_FILE_PATH} ${OUTPUT_FILE_PATH}
        DEPENDS ${TEST_FILE_PATH} ${UNITY_CONFIG_FILE}
        OUTPUT ${OUTPUT_FILE_PATH}
        WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}
    )
    # add the generated file to build
    target_sources(app PRIVATE ${OUTPUT_FILE_PATH})

    message(STATUS "Generating test runner ${OUTPUT_FILE_PATH} for
${TEST_FILE_PATH}")
endfunction()
```

And the Kconfig to enable unity:

modules/unity/Kconfig

```
config UNITY
    bool "Use Unity unit test framework"
    select TEST
    depends on !ZTEST
    help
        This enables unity test framework which you should use instead of ZTEST.
```

Since unity replaces ZTest, we want to make sure that ZTest is not being used.

Whew. That was a lot of code. This is what we need in order to generate and compile the test runners on Zephyr.

We can now modify our test CMakeLists.txt file to use unity:

tests/lib/example/example/CMakeLists.txt

```
if(CONFIG_CMOCK)
        unity_generate_test_runner(src/unit.c)
else()
        unity_generate_test_runner(src/integration.c)
        target_sources(app PRIVATE src/integration.c)
endif()
```

It's very convenient to use the same executable for both unit and integration tests. We'll cover why a bit later in this training.

We can now simply add test functions to src/integration.c file and the process will take care of generating the test runner and running them (we don't need to define a main function for our test):

```
void test_something_cool(void){
        TEST_ASSERT_EQUAL(0, 1);
}
```

Build and run the test:

```
west build -b native_posix tests/lib/example/example/ -t run
*** Booting Zephyr OS build zephyr-v3.0.0  ***
Parsing command line arguments
src/integration.c:12:test_something_cool:FAIL: Expected 0 Was 1

1 Tests 1 Failures 0 Ignored
FAIL
PROJECT EXECUTION FAILED
```

It is important to call "posix_exit" explicitly in our teardown because by default, zephyr doesn't exit the executable. Instead the program just hangs there - so we need to explicitly exit the process if we are running on native_posix.

Adding CMock

- Script to cleanup headers
- Script for dumping function names

Adding CMock is a bit more involved because we need to add scripts to do a few more things:

- **Clean up header files from things like 'static inline'** - this is necessary to ensure clean parsing by CMock ruby scripts.
- **Dump all functions from the header** - so that we can add linker options to redirect mocked methods to our wrappers across the whole application (super powerful!)

The second point is the easier one to solve. We can just use python libclang bindings to parse our header and dump all the function names:

```
import re
import argparse
import clang.cindex
from clang.cindex import CursorKind
import sys

# These are blacklisted because they are used early during init
# meaning that nothing is initialized yet - so we don't wrap them
blacklist = [
    "__errno_location",
    "k_work_queue_start",
    "z_init_static_threads",
    "k_sched_lock",
    "k_sched_unlock"
]

def dump_functions(in_file, out_file):
  with open(out_file, 'w') as f_out:
    index = clang.cindex.Index.create()
    tu = index.parse(in_file)
    for c in tu.cursor.walk_preorder():
      if c.kind == CursorKind.FUNCTION_DECL:
        if(c.spelling not in blacklist):
          f_out.write(c.spelling + "\n")

if __name__ == "__main__":
  parser = argparse.ArgumentParser()
  parser.add_argument("-i", "--input", type=str,
                      help="Input header to parse", required=True)
  parser.add_argument("-o", "--output", type=str,
                      help="Output file for function list", required=True)
  args = parser.parse_args()

  dump_functions(args.input, args.output)
```

The tricky part when mocking things with CMock on any RTOS is that we are building the whole system - kernel and all. This is one single executable. So if we mock some function, it also means that the function will be mocked for the kernel itself. If the kernel uses the mocked function it will simply never get to main because an important part of the system will be mocked.

This puts restrictions on what we can and can not mock. Luckily we will always know that something hasn't been mocked because if we don't add expectations to our test code then the test will fail. This gives us opportunity to add such functions to our "blacklist" which contains functions that we exclude from mocking at all times.

The output of the above script is a file containing list of all functions in the header file supplied as "input" parameter.

The next thing we need to do is generate a cleaned up version of our header file.

This can be done by filtering our header file in python. Things we need to filter out:

- **Static inline** - and variations thereof that are still being used in Zephyr.
- **Rewrite extern functions without extern** - so that they are just normal definitions (you should never use externs in your ordinary code!)
- **Delete comments** - to avoid feeding mock generator with unnecessary data.
- **Remove syscalls and syscall includes** - these have duplicate definitions of existing functions.
- *Add *_wrap* prefix to all functions and generate the wrap header.
- **And probably a whole lot more** - it is impossible to say which headers will not pass this process - you'll have to try generating mocks for the headers that you are using and then if you run into problems, add appropriate cleanup code.

This is a lot of stuff to be done to our headers before we can even start generating mocks! We could probably use libclang for this - but the biggest problem is that we have relatively few things to rewrite and relatively complex headers. So my current solution uses regex replacement instead.

```
import re
import argparse

def gen_headers(header, output, output_wrap):
  with open(header) as header_file:
    content = header_file.read()

  # Change static inline declarations to normal declarations
  pattern = re.compile(

r'(?:__deprecated\s+|static\s+inline\s+)?(?:static\s+inline\s+|static\s+ALWAYS_INLINE
\s+|__deprecated\s+)'
    r'((?:\w+[*\s]+)+\w+?\(.*?\))\n\{.+?\n\}',
    re.M | re.S)
  content = pattern.sub(r"\1;", content)

  # Remove externs, but only function externs (there are extern variables as well)
  pattern = re.compile(
    r'extern\s+((?:\w+[*\s]+)+\w+?\(.*?\);)',
    re.M | re.S)
  content = pattern.sub(r"\1", content)

  # Delete anything that matches these patterns completely
  patterns = [
    r'/\*(' # group match for 'any number of'
    r'[^*]|' # any number of non '*'
```

```
    r'[\r\n]|' # any number of return or new lines
    r'(\*+([^*/]|[\r\n]))' # any number of ('*' followed by not '*/') or (newlines)
    r')*' # end group for any number of
    r'\*+/', # ending with any number of '*' followed by slash '/'
  ]
  for pattern in patterns:
    pattern = re.compile(pattern, re.M | re.S)
    content = pattern.sub(r"", content)

  # Single line patterns
  patterns = [
    # Remove C++ comments
    r'//.*[\r\n]',
    # Remove syscalls because we will have duplicate declarations there
    r'#include <syscalls/\w+?.h>',
    # Remove attributes that mess up mock generation
    r'__syscall\s+',
    r'FUNC_NORETURN\s+',
    r'static\s+inline\s+',
    r'static\s+ALWAYS_INLINE\s+',
    r'__STATIC_INLINE\s+',
    r'inline\s+static\s+',
    r'__attribute_const__\s+',
    r'__deprecated\s+'
  ]
  for pattern in patterns:
    pattern = re.compile(pattern)
    content = pattern.sub(r"", content)

  # Treat as always enabled because otherwise futex mocks fail to compile
  pattern = re.compile(
    r'ifdef CONFIG_USERSPACE\s+',
    re.M | re.S)
  content = pattern.sub("if 1\n", content)

  with open(output, 'w') as output_file:
    output_file.write(content)

  # Prefix all function names with __wrap_ prefix
  pattern = re.compile(
      r"^\s*((?:\w+[*\s]+)+)(\w+?\s*\([^\\{}#]*?\)\s*;)", re.M)
  content = pattern.sub(r"\n\1__wrap_\2", content)

  with open(output_wrap, 'w') as output_file:
      output_file.write(content)
```

(for explanation of the regular expression flags have a look here)

This basically converts code like this (not valid code - just part of the automated test):

```
/**
 * This is a comment
 **/
extern void k_thread_foreach_unlocked(k_thread_user_cb_t user_cb, void *user_data);
extern FUNC_NORETURN void k_thread_user_mode_enter(k_thread_entry_t entry, void *p1,
void *p2,
                                                   void *p3);
```

```
extern void k_sys_runtime_stats_disable(void);

static inline __deprecated int foo(int a)
{
        return 0;
}
static inline int foo(int a)
{
        return 0;
}
static ALWAYS_INLINE int foo(int a)
{
        return 0;
}
```

Into this:

```
void k_thread_foreach_unlocked(k_thread_user_cb_t user_cb, void *user_data);
void k_thread_user_mode_enter(k_thread_entry_t entry, void *p1, void *p2,
                                                      void *p3);
void k_sys_runtime_stats_disable(void);

int foo(int a);
int foo(int a);
int foo(int a);
```

And this:

```
void __wrap_k_thread_foreach_unlocked(k_thread_user_cb_t user_cb, void *user_data);

void __wrap_k_thread_user_mode_enter(k_thread_entry_t entry, void *p1, void *p2,
                                                      void *p3);

void __wrap_k_sys_runtime_stats_disable(void);

int __wrap_foo(int a);

int __wrap_foo(int a);

int __wrap_foo(int a);
```

Finally we need to also tie this into our build process in Zephyr so that our headers get generated when building a test app:

modules/cmock/CMakeLists.txt

```
function(cmock_generate_mocks HEADER_PATH OUTPUT_PATH_PREFIX)
    # get the global cmock directory
    get_property(CMOCK_DIR GLOBAL PROPERTY CMOCK_DIR)

    # specify output as build/mocks
    set(CMOCK_OUTPUT_DIR ${APPLICATION_BINARY_DIR}/mocks)

    # specify output path for headers
    set(HEADER_OUTPUT_DIR "${CMOCK_OUTPUT_DIR}/${OUTPUT_PATH_PREFIX}")

    # make <prefix>/internal
    file(MAKE_DIRECTORY "${HEADER_OUTPUT_DIR}/internal")

    # get header filename
```

```
    get_filename_component(HEADER_FILE "${HEADER_PATH}" NAME)

    # modified header path
    set(MODIFIED_HEADER_PATH "${HEADER_OUTPUT_DIR}/${HEADER_FILE}")

    # wrapper header path
    set(WRAP_HEADER_PATH "${HEADER_OUTPUT_DIR}/internal/${HEADER_FILE}")

    # prepare headers
    cmock_gen_headers(${HEADER_PATH} ${MODIFIED_HEADER_PATH} ${WRAP_HEADER_PATH})
    cmock_gen_mocks(${WRAP_HEADER_PATH} ${HEADER_OUTPUT_DIR})

    # add linker options to wrap the functions
    cmock_add_linker_wraps(${MODIFIED_HEADER_PATH})

    # add the output directory to include path before all other paths
    target_include_directories(app BEFORE PRIVATE ${CMOCK_OUTPUT_DIR}/)

    message(STATUS "Generating mocks for ${HEADER_PATH}")
endfunction()
```

We now need to define a few functions:

- **cmock_gen_headers** - this will call our python script to generate headers.
- **cmock_gen_mocks** - this will actually invoke cmock ruby script and generate the mock sources.
- **cmock_add_linker_wraps** - this will add linker flags to wrap every single mocked function.

```
function(cmock_gen_headers IN_HEADER OUT_HEADER WRAP_HEADER)
    get_property(PROJECT_BASE GLOBAL PROPERTY PROJECT_BASE)
    execute_process(
        COMMAND
            ${PYTHON_EXECUTABLE}
            ${PROJECT_BASE}/modules/cmock/gen_headers.py
            --input ${IN_HEADER}
            --output ${OUT_HEADER}
            --wrap ${WRAP_HEADER}
        WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}
        RESULT_VARIABLE RETURN_CODE
        OUTPUT_VARIABLE SCRIPT_OUTPUT
    )

    if(NOT ${op_result} EQUAL 0)
        message(SEND_ERROR "${SCRIPT_OUTPUT}")
        message(FATAL_ERROR "Error parsing header ${IN_HEADER}")
    endif()
endfunction()
```

Now we need to generate mocks using the cmock utility:

```
function(cmock_gen_mocks HEADER_FILE OUTPUT_DIRECTORY)
    get_property(CMOCK_DIR GLOBAL PROPERTY CMOCK_DIR)
    # configure prefix for mock file
    set(MOCK_FILE_PREFIX mock_)
    # get filename without extension
    get_filename_component(FILE_NAME "${HEADER_FILE}" NAME_WE)
```

```
    # compute output file name
    set(MOCK_FILE ${OUTPUT_DIRECTORY}/${MOCK_FILE_PREFIX}${FILE_NAME}.c)
    # create the output directory
    file(MAKE_DIRECTORY "${OUTPUT_DIRECTORY}")

    add_custom_command(
        OUTPUT ${MOCK_FILE}
        COMMAND
            ${RUBY_PATH} ${CMOCK_DIR}/lib/cmock.rb
            --mock_prefix=${MOCK_FILE_PREFIX}
            --mock_path=${OUTPUT_DIRECTORY}
            -o${CMOCK_CONFIG_FILE}
            ${HEADER_FILE}
        DEPENDS ${HEADER_FILE} ${CMOCK_CONFIG_FILE}
        WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}
    )
    target_sources(app PRIVATE ${MOCK_FILE})
endfunction()
```

And finally we add linker flags to redirect our mocked functions to the generated code so that we can control execution of our program:

```
function(cmock_add_linker_wrap_functions FUNCTION_LIST_FILE)
    file(STRINGS ${FUNCTION_LIST_FILE} CONTENT)
    if(CONTENT)
        set(LINKER_OPTS "-Wl")
    endif()
    foreach(FUNC_NAME ${CONTENT})
        set(LINKER_OPTS "${LINKER_OPTS},--wrap=${FUNC_NAME}")
    endforeach()
    zephyr_link_libraries(${LINKER_OPTS})
endfunction()

function(cmock_add_linker_wraps HEADER_PATH)
    get_property(PROJECT_BASE GLOBAL PROPERTY PROJECT_BASE)
    set(FUNCTION_LIST_FILE "${HEADER_PATH}.flist")

    execute_process(
        COMMAND
            ${PYTHON_EXECUTABLE}
            ${PROJECT_BASE}/modules/cmock/gen_function_list.py
            --input ${HEADER_PATH}
            --output ${FUNCTION_LIST_FILE}
        WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}
        RESULT_VARIABLE RETURN_CODE
        OUTPUT_VARIABLE SCRIPT_OUTPUT)

    if(NOT ${RETURN_CODE} EQUAL 0)
        message(SEND_ERROR "${RETURN_CODE}")
        message(FATAL_ERROR "Error generating function list from ${HEADER_PATH}")
    endif()
    cmock_add_linker_wrap_functions(${FUNCTION_LIST_FILE})
endfunction()
```

How was this for a massive body of work just to get unit testing working (**it's completely worth it because the power it gives to verify everything is insane**). Traditionally, unit testing driver code fully was a puzzling task - with CMock unit testing is easy. For the first time we can

check 100% of the logic that we write no matter where in our software stack our code will run!

Testing code with CMock

- Generate mocks using CMake commands
- Setup expectations
- Run code
- CMock automatically verifies the expectaions and call order

We can now unit test our modules with CMock.

We have our example library:

lib/example/example.c

```
int example_object_init(struct example_object *self)
{
    if (!self)
        return -EINVAL;
    memset(self, 0, sizeof(*self));
    printk("Example object initialized!\n");

    k_mutex_init(&self->mx); (1)
    return 0;
}
```

(1) this function is defined in kernel.h

We configure our build to mock everything in kernel.h:

tests/lib/example/example/CMakeLists.txt

```
if(CONFIG_CMOCK)
  unity_generate_test_runner(src/unit.c)
  cmock_generate_mocks(${ZEPHYR_BASE}/include/kernel.h .)
  target_sources(app PRIVATE src/unit.c)
else()
  unity_generate_test_runner(src/integration.c)
  target_sources(app PRIVATE src/integration.c)
endif()
```

Now if we write a test in our unit.c file that calls example_object_init:

```
void test_example(void)
{
    struct example_object ex;
    TEST_ASSERT_EQUAL(0, example_object_init(&ex));
}
```

And run it:

```
west build -p -b native_posix tests/lib/example/example/ -t run -- -DCONFIG_CMOCK=y
```

We are going to get an error from CMock:

```
src/unit.c:13:test_example:FAIL:Function __wrap_k_mutex_init.  Called more times than
expected.

1 Tests 1 Failures 0 Ignored
```

This failed test tells us that we have failed to set expectations for our mocked k_mutex_init method. Let's do it:

```
#include <unity.h>
#include <example/example.h>
```

```
#include <mock_kernel.h>

// this is a unit test so we will include the file directly
// this is the only place where we do this - it is the optimal solution.
#include "../../../../../lib/example/example.c"

void test_example(void)
{
    struct example_object ex;
    __wrap_k_mutex_init_ExpectAndReturn(&ex.mx, 0);
    TEST_ASSERT_EQUAL(0, example_object_init(&ex));
}
```

Now the test succeeds!

Including the source file in unit test

You may have noticed above that we are including the source file itself into the unit test. This is generally considered a bad practice in C and many developers frown on it when they first see it. However, unit testing is a very special case where it is not only ok but actually the only possible solution.

When we are writing a unit test, the goal is to test every single path through our code. This means we want to individually test internal methods inside the file being tested. When we include the file into our test, it is equivalent of appending the test code to the end of the source file - this is exactly what we want. This gives us full access to everything inside the file being tested.

Testing drivers

When you are testing drivers, you need to make sure that the driver does not initialize automatically. You can achieve this by creating a file that you include before your driver under test and undefine the driver init macro inside it:

```
#undef DT_INST_FOREACH_STATUS_OKAY
#define DT_INST_FOREACH_STATUS_OKAY(macro)
```

This way you will not run into the problem of your unit test crashing when your driver initializes before CMock has a chance to start.

System testing using Renode

- Simulate production firmware
- Use robot framework to describe user scenarios
- Verify firmware behavior in simulation

The last piece of the puzzle that we need to get working is the renode testing. Renode is a simulator that can be used with Robot framework. The fantastic advantage of renode is that it can be seamlessly integrated into the continuous testing process and allow us to verify every single user requirement automatically. This way we can focus the team on implementing user requirements during each sprint and have a way of verifying that all user requirements are being met continuously.

Renode is already partially supported in Zephyr (thanks to Antmicro) and all we have to do to start using it is enable it for our custom board.

We add a board specific script that Zephyr can run when we execute "run" command:

boards/arm/custom_board/support/board.resc

```
include @scripts/single-node/stm32f4_discovery.resc

showAnalyzer sysbus.usart2

macro reset
"""
        sysbus LoadELF $bin
"""

runMacro $reset
```

Then we enable renode in the board Makefile:

```
# specify which platforms are supported
set(SUPPORTED_EMU_PLATFORMS renode)
# specify default platform for "run" command
set(EMU_PLATFORM renode)
# specify path to default script to run simulation
set(RENODE_SCRIPT ${CMAKE_CURRENT_LIST_DIR}/support/board.resc)
```

This instructs zephyr where to find our script.

Now we can run our application using renode:

```
west build -p -b custom_board tests/lib/example/example/ -t run
15:17:12.7989 [INFO] Loaded monitor commands from: /opt/renode/scripts/monitor.py
15:17:12.8540 [INFO] Including script: /data/12/personal/zephyr-workspace/zephyr-
getting-s
tarted/boards/arm/custom_board/support/board.resc
15:17:12.8848 [INFO] System bus created.
15:17:14.3525 [INFO] sysbus: Loaded SVD: /tmp/renode-613032/ac9bb70e-96f4-4291-a35f-
2130db
6fa785.tmp. Name: STM32F40x. Description: STM32F40x.
15:17:14.6231 [INFO] sysbus: Loading segment of 26976 bytes length at 0x8000000.
15:17:14.6347 [INFO] sysbus: Loading segment of 112 bytes length at 0x8006960.
15:17:14.6348 [INFO] sysbus: Loading segment of 5552 bytes length at 0x20000070.
15:17:14.6825 [INFO] sysbus: Loading segment of 26976 bytes length at 0x8000000.
15:17:14.6826 [INFO] sysbus: Loading segment of 112 bytes length at 0x8006960.
```

```
15:17:14.6827 [INFO] sysbus: Loading segment of 5552 bytes length at 0x20000070.
15:17:14.7536 [INFO] cpu: Guessing VectorTableOffset value to be 0x8000000.
15:17:14.7557 [INFO] cpu: Setting initial values: PC = 0x8001495, SP = 0x20000860.
15:17:14.7578 [INFO] STM32F4_Discovery: Machine started.
15:17:14.8559 [INFO] usart2: [host: 0.18s (+0.18s)|virt: 0.1ms (+0.1ms)] *** Booting
Zephyr OS build zephyr-v3.0.0  ***
```

But this is not enough. We also want to run robot scripts against our compiled firmware. To do that we need to start renode from robot framework and initialize everything there instead.

When you introduce renode there are a couple of things to keep in mind:

- **Renode scripts work with final compiled artifacts** - this is no longer part of the build process - rather it runs after everything has been built.
- **Renode scripts are not run from west** - they are run using "robot-test" special script.

In order to make this process effective, we need to keep our build output paths standardized. If you have followed this whole training so far, then your paths will be all correct.

We can verify that our test firmware boots by running the following script:

```
*** Settings ***
Suite Setup                   Setup
Suite Teardown                Teardown
Test Setup                    Reset Emulation
Test Teardown                 Test Teardown
Resource                      ${RENODEKEYWORDS}
Variables                     ${CURDIR}/../../../scrum/variables.py

*** Variables ***
${UART}                       sysbus.usart2

*** Test Cases ***
Run Shell Sample
    Execute Command           set bin @${PROJECT_ROOT}/build-
all/custom_board/shell/apps.shell.release/zephyr/zephyr.elf
    Execute Command           include @scripts/single-node/stm32f4_discovery.resc

    Execute Command           showAnalyzer ${UART}
    Create Terminal Tester    ${UART}

    Start Emulation

    Wait For Line On Uart     Booting Zephyr OS
```

Here I include the variables file from the scrum folder, but notice that I'm not placing this script in scrum folder. This is because we use different commands to run this script. this scripts need to be run using the robot framework test runner instead of robot:

```
$ renode-test renode/boards/arm/custom_board.robot
Preparing suites
Started Renode instance on port 9999; pid 618707
Starting suites
Running renode/boards/arm/custom_board.robot
+++++ Starting test 'custom_board.Run Shell Sample'
+++++ Finished test 'custom_board.Run Shell Sample' in 2.39 seconds with status OK
Cleaning up suites
Closing Renode pid 618707
```

```
Aggregating all robot results
```

Nice!

Lastly we need to add this to our CI process.

I have chosen to do so in the check script right before running the scrum tests:

```
for FILE in find renode -name "*.robot"; do
    renode-test $FILE || {
        echo "Renode test failed ($FILE)"
        exit 1
    }
done
```

renode-test doesn't support running all files in a directory - so we have to use this more explicit approach.

Adding code coverage check

- Filter out project files
- Generate report only for project files
- Fail pipeline if coverage below threshold

The most important step of this module is to add coverage check to your automatic build process so that we place a hard requirement on developers to test their code fully.

You only need coverage checks for unit tests - these are the tests that focus on checking all internal logic of a module. These tests MUST have full coverage (you simply can not afford not to do this).

Integration tests don't need to be part of coverage - rather they should be tied to requirements. I haven't included this into this module because then this module would be huge - but what we did do is add command line support which allows us to pass to unity a name pattern of the tests we want to run. This means that you can call your native posix integration test (after it is built and placed into a well defined artifact tree) from renode and make sure that keywords are implemented inside the integration test.

With system tests you don't need coverage either because they check how well the final application satisfies specific use case scenarios. Coverage doesn't tell us much about that. So don't use coverage for that.

Here is what we want:

- **100% line coverage** - this is easily achievable with CMock and unit testing.
- **100% branch coverage** - this is not easily achievable on Zephyr and requires disabling logging for example (because log macros contain branches we can not test). So we have to settle for "maximum coverage" - but we still have to check it in our build process and fail if it isn't above a specific value.

When we run twister with --coverage flag and it runs all the tests that are marked for being run on native_posix platform, coverage is collected and accumulated for all runs in the build-all/coverage.info file.

All we have to do is extract the data that we want from this file (for example we don't care about the coverage of zephyr files - only our own) and verify that we have coverage above threshold only for the files we are interested in.

There is a caveat however, if a source file is not included into any build process, it will not appear in coverage - so lastly we also will need to compare our list of source files that we have in our project to the list of files in the coverage report. If any file is missing then we fail our pipeline as well.

First we need to extend our build script to not only build everything, but also include coverage. For this we need to enable coverage option and also set the coverage platform. The --coverage-platform specifies that we will only be generating coverage from tests that we actually run locally (built for native_posix):

scripts/build

```
$ROOT/../zephyr/scripts/twister \
      --integration \
```

```
+     --coverage \
+     --coverage-platform native_posix \
      -c \
...
```

If we now run our build script, we can view the coverage generated:

```
$ ./scripts/build
$ lcov --list build-all/coverage.info --rc lcov_branch_coverage=1
modules/test/cmock/src/cmock.c                 |35.0%    60|40.0%  10|25.0%   36
modules/test/cmock/vendor/unity/src/unity.c    |12.7%   810|28.3%  46| 5.2%  580
zephyr-getting-started...r/runner_integration.c|63.3%    49|80.0%  10|40.0%   10
zephyr-getting-started...it/mocks/mock_kernel.c| 0.1% 25071| 0.1%  2k| 0.1% 9458
zephyr-getting-started...t/runner/runner_unit.c|63.3%    49|80.0%  10|40.0%   10
zephyr-getting-started/lib/example/example.c   |85.7%     7| 100%   1|50.0%    2
zephyr-getting-started/modules/unity/main.c    |80.0%     5| 100%   1|    -    0
zephyr-getting-started/modules/unity/teardown.c|85.7%     7| 100%   2|50.0%    2
```

This output contains coverage for all files included in all our builds (cumulative coverage) as absolute paths.

In our CI we only care about our current project so we will filter this coverage data to only include information about modules inside a handful of predefined directories.

```
$ lcov --extract build-all/coverage.info "$PWD/lib/*" "$PWD/drivers/*" --exclude
"*runner*" --rc lcov_branch_coverage=1 -o build-all/coverage-filtered.info
$ lcov --rc lcov_branch_coverage=1 --list-full-path --list build-all/coverage-
filtered.info
             |Lines       |Functions  |Branches
Filename     |Rate     Num|Rate    Num|Rate     Num
================================================
[lib/example/]
example.c    |85.7%      7| 100%     1|50.0%      2
================================================
       Total:|85.7%      7| 100%     1|50.0%      2
```

NOTE

> We use --list-full-path because then we can use "awk" command to filter the list and compare it with actual files that we have present in our directories.

Note that our example_driver.c file is not present here. This is because we currently haven't written a test for it (you can do it as an exercise). If we list all our C files then we get:

```
$ find drivers lib -name "*.c"
drivers/example/example_driver.c
lib/example/example.c
```

We thus need to also generate an error if files that we find in the folders are not present in the lcov output.

We can do it by comparing list of files to list of files in coverage report based at current directory:

```
diff \
    <(find lib drivers -name "*.c" | sort)
    <(lcov \
        --rc lcov_branch_coverage=1 \
        --list-full-path \
        --list build-all/coverage-project.info | \
            grep $PWD |
```

```
            awk -F '|' '{print $1;}' |
            xargs realpath --relative-to=$PWD |
            sort
    )
```

The bash syntax "diff <(a) <(b)" means that we treat outputs of two commands as files that are passed to diff. Thus in our case this means simply "diff outputs of two commands".

Then we take output of lcov and search inside it for lines starting with our project directory - this removes all other lines that we don't need.

Then we grab the first path of the output (the table is separated by '|') using awk and finally we translate the paths such that they are based on current project directory.

Lastly the output is sorted.

If all files are in coverage then the lists of files are identical. Otherwise the diff command returns an error.

When it comes to the coverage check, we can extract coverage data from the same command:

```
#!/bin/bash

ROOT="$(realpath $(dirname $BASH_SOURCE)/..)"

set -e

BUILD_DIR=$1

if [[ ! -d $BUILD_DIR ]]; then
    echo "Build directory $BUILD_DIR does not exist"
    echo "run ./scripts/build"
    exit 1
fi

# Filter coverage data
lcov \
    --extract $BUILD_DIR/coverage.info \
    "$PWD/include/*" \
    "$PWD/drivers/*" \
    "$PWD/lib/*" \
    --rc lcov_branch_coverage=1 \
    -o $BUILD_DIR/coverage-project.info

# List all coverage data we have
lcov \
    --rc lcov_branch_coverage=1 \
    --list $BUILD_DIR/coverage-project.info

# Generate html report
genhtml $BUILD_DIR/coverage-project.info \
    --output-directory $BUILD_DIR/coverage-project \
    --rc lcov_branch_coverage=1

echo "HTML coverage report is in $BUILD_DIR/coverage-project"

# Check that we have full coverage
BRANCH_COVERAGE_TH=100
```

```
LINE_COVERAGE_TH=100

BRANCH_COVERAGE=$(\
    lcov \
    --rc lcov_branch_coverage=1 \
    --summary $BUILD_DIR/coverage-project.info | \
    grep "branches" | \
    sed -e 's/[[:space:]]*branches\.*: //g' | \
    sed -e 's/%.*//g' \
)
LINE_COVERAGE=$( \
    lcov \
    --rc lcov_branch_coverage=1 \
    --summary $BUILD_DIR/coverage-project.info | \
    grep "lines" | \
    sed -e 's/[[:space:]]*lines\.*: //g' | \
    sed -e 's/%.*//g' \
)

if [[ $(echo "$BRANCH_COVERAGE_TH<=$BRANCH_COVERAGE" | bc) = "0" ]]; then
    echo "Error branch coverage - expected: $BRANCH_COVERAGE_TH, got:
$BRANCH_COVERAGE)"
    exit 1
fi

if [[ $(echo "$LINE_COVERAGE_TH<=$LINE_COVERAGE" | bc) = "0" ]]; then
    echo "Error line coverage - expected: $LINE_COVERAGE_TH, got: $LINE_COVERAGE)"
    exit 1
fi
```

Additional details

- Ensure each driver has a test

Ensuring each driver has a test

We can ensure that each driver in the drivers directory has a corresponding test by simply adding a little code to the driver directory check in infrastructure/directory-structure.robot:

```
FOR  ${DRIVER}  IN  @{DRIVERS}
    # make sure there is a test for each driver
    ${PATH}  ${FILE} =  Split Path  ${DRIVER}
    ${NAME}  ${EXT} =  Split Extension  ${FILE}
    Directory Should Exist  ${ROOT_DIR}/tests/drivers/${DRIVER_TYPE}/${NAME}/
END
```

Pitfalls

- Using too much "odd" functionality
- Using static inline
- Not catching files that have no coverage at all
- Renode bugs

The biggest pitfall with tests is simply not implementing support for ALL the test kinds. You really need all three kinds of tests - unit, integration and system.

Zephyr by default only has decent support for integration tests - and even at that it is using its own rather primitive ZTest framework for it. To do proper embedded development you also need support for unit tests and system tests.

Other pitfalls include:

- **Not updating the gen_headers.py file** - this file will need to be updated as you build your project and more things are likely to need to be filtered. When you get a compilation error related to mocks, this is the first file you should check.
- **Renode bugs** - renode supports massive amount of hardware - but bugs still occur. Recently I had a problem running a GPIO test because of how renode handled bit sets and resets inside the STM32 GPIO emulation. That was a bug in renode and it made the test fail. Be aware that you may need to patch renode occasionally (it's quite easy though compared to doing the same to qemu)
- **Missed coverage** - if a file is not used in any test, it will not be part of any coverage reports. That means you may have misleading "100% coverage" readings.

Success metrics

- Full code coverage using unit tests
- Good functional coverage through integration tests
- Good user requirement coverage through simulation tests

This module has established a whole new standard for success metrics. We are now have:

- **Enforced code coverage** - no sources can be added without also adding tests.
- **Robot framework to the rescue** - we have so far encoded most of our success metrics into robot language. This trend must continue so that we can say "if robot tests pass we are doing great".

There is a huge benefit of scripting all processes using a language as robot. You can apply it not just to testing things like this in embedded - but to any area where operational productivity is very important.

Summary

In this training we have covered a huge bulk of materials on testing. You probably have to go back and reread several sections (and most importantly - apply!).

Here is a quick summary of what we have covered:

- **Adding unity support** - so that we can write all our tests against standardized set of assert macros.
- **Added cmock support** - so that we can mock anything in our software and test things in isolation.
- **Added renode testing** - so that we can automate user requirements verification.
- **Added test coverage checking** - so that we never let a merge request be merged unless all code has been tested for logical correctness.

What to do next

- Get consulting here: https://swedishembedded.com/go

If you want to maximize your team's operational effectiveness, chances are that a lot of content in this training is highly applicable for solving problems that you face. There is more - much more.

Swedish Embedded Group would be happy to help you see new ways to do things better. With long experience in embedded systems and a unique ability to see innovative ways of automating your processes, we can absolutely guarantee you that if you are working with firmware development of any kind then there will be something important that we can help you improve.

The only way to find out is to book a free conversation online. I have included in this book a link to a calendar. When you go to the calendar, you will be able to pick a time that works best for you from a list of available spots. Once you have booked a time, you'll get an email with a zoom link that you can use to connect to the meeting.

Click here to book a call now

```
Author: Martin Schröder, 20 Jun 2022
Email: martin.schroder@swedishembedded.com
Training: Zephyr Getting Started Training
Module: 5
```

Chapter 6

Release Generation

► https://www.youtube.com/watch?v=9E6XTUdYVVQ *(YouTube video)*

Recap

- We have a repository
- We have CI setup and working with docker image
- We have an organized directory structure
- We are able to generate docs efficiently
- We have testing and code coverage

At this point we are almost done. The repository is in good shape, we have an excellent CI workflow and we have code coverage checking.

Problems we have

- No release generation
- No software bom tracking

In our current infrastructure we have created a system where the master branch is always ready for release. Each time a new commit is added to the master branch, we build all applications in our project.

This creates a problem: once we have flashed a release onto any device in the field - how do we actually know what went into it? We can of course reference the git checksum of the commit from which the release has been built - but even if we check out that release at a later point and try to rebuild it - we have no way to know whether it is exactly the same or not. In fact, chances are that check sums of the built archives will be different - it is enough that only a single bit differs and the checksum will be completely different.

So we need a better way to compare two releases - and more importantly be able to track each component to the place where it was last changed in git.

We can do this using SPDX software bill of materials (Software BoM). Zephyr already supports generating the software BoM so all we have to do is build our application with software BoM generation enabled.

The software BoM gives us information about every single software component that is part of the final firmware executable:

```
FileName: ./zephyr/subsys/logging/log_backend_uart.c
SPDXID: SPDXRef-File-log-backend-uart.c
FileChecksum: SHA1: 96902cda78e83c1f5a8699b1af2d1591f196339f
FileChecksum: SHA256:
d678cff866dbba24dacebf327c53c43faf39dcb36a51015f8889d83eaf0f2e08
LicenseConcluded: Apache-2.0
LicenseInfoInFile: Apache-2.0
FileCopyrightText: NOASSERTION
```

The traceability that we get with a software bom goes far beyond anything we can get with just version numbers and tracking which git commit we have built from.

- **Compare two releases** - we can for example compare two releases without having access to the git repository.
- **Track file version to git commit** - we can also search git for the exact SHA1 hash of the file and find the last commit where that file was changed to the new version - giving us the last changes made to that file.
- **Inspect content of a release** - we can see exactly what files were compiled in - few other methods give us this level of detail.

Essentially, the methods described in this module will give you complete traceability from a packaged release to every single file (and the corresponding changes made to that file) that went into it.

Solution

- Adopt semantic versioning
- Add software bom generation
- Add packaging of releases

The first thing we are going to do is describe briefly the "semantic versioning scheme" and how it simplifies versioning of software.

Then we are going to configure software bom generation and finally we will package the release into an archive containin everything that is needed.

We will also add:

- **Application version numbers** - this will give us a global version number for all applications in our project.
- **Per file versioning** - this will give us a list of source files with corresponding git check sums that will go into a release (and allow us to trace files back from release to a specific git commit where the file was last changed).
- **Package everything into a single archive** - one archive per application and each archive will contain all configurations built for that application.
- **Save artifacts in CI** - so that the release can be downloaded at any time.

Let's define the scope of what we will do. We do this as a robot framework script:

```
Release of ${APP} is built for ${BOARD}
        ${NAME} =  Replace String  ${APP}  /  .
        Set Test Variable  ${RELEASE_BUILD_DIR}  build-release/${BOARD}/${APP}/
        File Should Exist  ${RELEASES}/${NAME}-${BOARD}-${VERSION}.tar.gz
        App release contains all necessary files

App release contains all necessary files
        File Should Exist  ${RELEASE_BUILD_DIR}/release/spdx/zephyr.spdx
        File Should Exist  ${RELEASE_BUILD_DIR}/release/spdx/app.spdx
        File Should Exist  ${RELEASE_BUILD_DIR}/release/spdx/build.spdx
        File Should Exist  ${RELEASE_BUILD_DIR}/release/zephyr.elf
        File Should Exist  ${RELEASE_BUILD_DIR}/release/zephyr.hex
        File Should Exist  ${RELEASE_BUILD_DIR}/release/reference.pdf
```

For the remainder of this module we will write CMake and shell scripts to automate the release process. We will use robot framework to verify that we have all necessary files built after the whole process has completed.

Semantic versioning

- Uses MAJOR.MINOR.PATCH pattern
- Versions below 1.0.0 may change arbitrarily
- Update MAJOR version when you make incompatible API changes
- Update MINOR version when you add functionality in a backwards compatible manner
- Update PATCH version when you make backwards compatible bug fixes
- More info at semver.org

You can read more about semantic versioning at http://semver.org

Adding project versioning

- Add verison to **`VERSION`** file
- Add template for header: **`version.h.in`**
- Add cmake file to parse **`VERSION`** file

Earlier in this training we have added a VERSION file, but we haven't added any code to generate version headers. This was because at the time we didn't have the application code and so there was no way of verifying our work fully. Now that we have an example application in our repository, we can add version file support and verify that our application prints the same version as the one we have in the version file.

To add version support to our project, we add two files:

- **version.h.in** which will serve as template for generating a header that can be included into our applications.
- **cmake/version.cmake** which will generate the version header from the version file inside our repository.

We can reuse the version template from Zephyr, but strip it down only to the basic three components and replace "KERNEL" with "PROJECT":

version.h.in

```
#ifndef _PROJECT_VERSION_H_
#define _PROJECT_VERSION_H_

#cmakedefine PROJECT_VERSION_CODE @PROJECT_VERSION_CODE@
#define PROJECT_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))

#define PROJECT_VERSION_NUMBER  @PROJECT_VERSION_NUMBER@
#define PROJECT_VERSION_MAJOR   @PROJECT_VERSION_MAJOR@
#define PROJECT_VERSION_MINOR   @PROJECT_VERSION_MINOR@
#define PROJECT_VERSION_PATCHLEVEL      @PROJECT_VERSION_PATCHLEVEL@
#define PROJECT_VERSION_STRING  @PROJECT_VERSION_STRING@

#define PROJECT_BUILD_VERSION          @PROJECT_BUILD_VERSION@

#endif
```

These defines will be visible in all of our code after including the generated header.

We can now add the CMake code that will convert our version file into both the CMake definitions of version codes and also into the generated header that we can use in our application.

```
get_property(PROJECT_BASE GLOBAL PROPERTY PROJECT_BASE)
include(${ZEPHYR_BASE}/cmake/hex.cmake)

if(NOT DEFINED PROJECT_BUILD_VERSION)
  find_package(Git QUIET)
  if(GIT_FOUND)
    execute_process(
      COMMAND ${GIT_EXECUTABLE} describe --abbrev=12 --always
      WORKING_DIRECTORY                 ${PROJECT_BASE}
      OUTPUT_VARIABLE                   PROJECT_BUILD_VERSION
```

```
      OUTPUT_STRIP_TRAILING_WHITESPACE
      ERROR_STRIP_TRAILING_WHITESPACE
      ERROR_VARIABLE                  stderr
      RESULT_VARIABLE                 return_code
    )
    if(return_code)
      message(STATUS "git describe failed: ${stderr}")
    elseif(NOT "${stderr}" STREQUAL "")
      message(STATUS "git describe warned: ${stderr}")
    endif()
  endif()
endif()

file(READ ${PROJECT_BASE}/VERSION ver)

string(REGEX MATCH "VERSION_MAJOR=([0-9]*)" _ ${ver})
set(PROJECT_VERSION_MAJOR ${CMAKE_MATCH_1})

string(REGEX MATCH "VERSION_MINOR=([0-9]*)" _ ${ver})
set(PROJECT_VERSION_MINOR ${CMAKE_MATCH_1})

string(REGEX MATCH "VERSION_PATCH=([0-9]*)" _ ${ver})
set(PROJECT_VERSION_PATCH ${CMAKE_MATCH_1})

set(PROJECT_VERSION
${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}.${PROJECT_VERSION_PATCH})

if(DEFINED PROJECT_BUILD_VERSION)
    set(PROJECT_BUILD_VERSION_STR ", build: ${PROJECT_BUILD_VERSION}")
endif()

message(STATUS "Building project version: ${PROJECT_VERSION}")

set(MAJOR ${PROJECT_VERSION_MAJOR}) # Temporary convenience variable
set(MINOR ${PROJECT_VERSION_MINOR}) # Temporary convenience variable
set(PATCH ${PROJECT_VERSION_PATCH}) # Temporary convenience variable

math(EXPR PROJECT_VERSION_NUMBER_INT "(${MAJOR} << 16) + (${MINOR} << 8)  +
(${PATCH})")
to_hex(${PROJECT_VERSION_NUMBER_INT} PROJECT_VERSION_NUMBER)

set(PROJECT_VERSION_STRING     "\"${PROJECT_VERSION}\"")
set(PROJECT_VERSION_CODE ${PROJECT_VERSION_NUMBER_INT})

configure_file(
    ${CMAKE_CURRENT_LIST_DIR}/version.h.in
    ${PROJECT_BINARY_DIR}/include/generated/project_version.h)

# Cleanup convenience variables
unset(MAJOR)
unset(MINOR)
unset(PATCH)
```

This creates following variables available to us globally:

```
PROJECT_VERSION: 0.0.1
PROJECT_VERSION_STRING: "0.0.1"
```

```
PROJECT_VERSION_MAJOR: 0
PROJECT_VERSION_MINOR: 0
PROJECT_VERSION_PATCH: 1
PROJECT_VERSION_NUMBER: 0x1
```

Software BOM

Software BoM is provided by the west "spdx" extension. Since this is an extension to west and is not integrated into twister, we can not just instruct twister to generate the BoM. What we have to do is rebuild the projects that we want to release using a series of west commands so that the software BoM is generated.

We are going to integreate these commands into our release build script. .scripts/build

```
west spdx --init -d $BUILD_DIRECTORY
west build -d $BUILD_DIRECTORY
west spdx -d $BUILD_DIRECTORY
```

Once the build process completes, we will have spdx files in "spdx" subdirectory of our build directory:

```
├──── app.spdx
├──── build.spdx
└──── zephyr.spdx
```

Packaging up a release

- Executable (.elf)
- Config (.config)
- Device tree used (.dts)
- SPDX bom
- Documentation

We can create a helper script that will build the release archive for us:

```
#!/bin/bash

ROOT="$(realpath $(dirname $BASH_SOURCE)/..)"
set -e

. ${ROOT}/../zephyr/zephyr-env.sh
. ${ROOT}/VERSION

# Parse arguments (standard way of doing it in bash)
while [[ $# -gt 0 ]]; do
    case $1 in
    -b|--board)
        PLATFORM="$2"
        shift
        shift
        ;;
    -s|--source)
        SOURCE_DIRECTORY="$2"
        shift
        shift
        ;;
    *)
        POS_ARGS+=("$1")
        shift
        ;;
    esac
done

if [[ $SOURCE_DIRECTORY = "" ]]; then
    echo "No source directory given";
    exit 1
fi

if [[ $PLATFORM = "" ]]; then
    echo "No platform given";
    exit 1
fi

# Create build directory
BUILD_DIRECTORY=build-release/${PLATFORM}/$(\
        realpath --relative-to=$ROOT $SOURCE_DIRECTORY)
mkdir -p $BUILD_DIRECTORY

# Get name of output archive
OUTPUT_ARCHIVE="$(echo \
```

```
    $(realpath \
    --relative-to=$ROOT/build-release/${PLATFORM} \
    $BUILD_DIRECTORY) |\
    tr "\/" "." \
    )-$(echo \
    ${PLATFORM} | tr '@' '_'\
    )-${PLATFORM}-${VERSION_MAJOR}.${VERSION_MINOR}.${VERSION_PATCH}"
OUTPUT_ARCHIVE="$ROOT/release/$OUTPUT_ARCHIVE.tar.gz"

mkdir -p $ROOT/release

# Build a release
west spdx --init -d $BUILD_DIRECTORY
west build -d $BUILD_DIRECTORY -b $PLATFORM -s $SOURCE_DIRECTORY
west spdx -d $BUILD_DIRECTORY

# Copy release files
pushd $BUILD_DIRECTORY
mkdir -p release
cp -r spdx release
cp -r zephyr/zephyr.* release
cp zephyr/.config release
cp $ROOT/doc/_build/latex/zephyr.pdf release/reference.pdf
# Pack release
tar -czf $OUTPUT_ARCHIVE release/*
popd

echo "Release archive created: $(realpath --relative-to=$ROOT $OUTPUT_ARCHIVE)"
```

We then add this script to our scripts/build script so that releases are built at the end (docs need to be built before so that reference manual can be included with the release):

```
# Build releases
./scripts/release -b custom_board@<version> -s apps/shell
```

Since our robot framework scripts always run last after all files have been created, the tests that we have written at the start will run and verify that our releases have been created (and contain all necessary files)

Saving artifacts to CI

- Add files you want to save under **`artifacts`** in **`.gitlab-ci.yml`**

Lastly we need to make sure that our release archives are saved in CI. We can do this by specifying the artifact paths in our ".gitlab-ci.yml" file:

gitlab-ci.yml

```
merge_request_pipeline:
...
    artifacts:
        paths:
            # Save all releases
            - release/*
        expire_in: 2 yrs
        when: always
...
```

Once the pipeline builds, on the results page you will be able to see the artifacts under the "Job artifacts" section to the right.

Finding git commit from sha1 sum

- Get file sha1 from SPDX bom
- Search git history for file version with that hash

When we have the SPDX BoM as a starting point, one thing that we may want to do is to find out where a particular file was last changed. To do this we need to list all sha1 sums of a file in history and then search that list for our particular sha1 sum and output the git commit where it appears.

I wrote a simple script to do this:

scripts/spdx/find_git_commit_for_file_sha1

```
#!/bin/bash

usage(){
    echo "$0 -f <file> -s <sha1sum>"
}

while [[ $# -gt 0 ]]; do
    case $1 in
    -s|--sha1)
        SHA1SUM=$2
        shift
        shift
        ;;
    -f|--path)
        FILE="$2"
        shift
        shift
        ;;
    *)
        usage
        exit 1
        ;;
    esac
done

if [[ "$FILE" = "" ]] || [[ "$SHA1SUM" = "" ]]; then
    usage
    exit 1
fi

git log --format=%H \
    | xargs -Iz sh -c \
        "echo -n \"z \"; git show z:$FILE | sha1sum" 2>/dev/null \
    | grep -m1 $SHA1SUM \
    | cut -d " " -f 1 \
    | xargs -Iz git log z -1 --format=%H
```

Test it:

```
$ cat <app.spdx file>
...
FileName: ./src/main.c
SPDXID: SPDXRef-File-main.c
```

```
FileChecksum: SHA1: 6f32112d6a0ae1d0775df25e8301b89ad965f911
...
$ ./scripts/spdx/find_git_commit_for_file_sha1 -f apps/shell/src/main.c -s
6f32112d6a0ae1d0775df25e8301b89ad965f911
7dd0803d41d05fcc7ec2dc2dce8358e1e15a22d1
```

Pitfalls

- Too many firmware configurations
- Not making default build the release build

The biggest pitfall is not building the firmware with the same configuration for the release package as we do through twister.

- **Default build should be release build** - The default build that we get when we run "west build" command should be the default release build. If you have other builds, you can configure them in the sample.yaml file of your application - but make sure you only have a single release build for each application.
- **Having multiple firmware binaries** - it is highly desirable to build only one binary as a final product and then do any customisations through dynamic configuration of the device. Having only one binary ensures that there is never a confusion as to what should be flashed where. Of course, if you have multiple versions of your board then you will have to build multiple binaries
- but these are already supported and will produce different final archive files.

Success metrics

- Trunk is always "releasable"
- A fully deployable release is built each time in CI
- Releases are reproducible from git history

At this point you should have a fully scripted system for producing a release archive. You should have:

- **Compliance checks** - that check for obvious oversights in the code that you are committing.
- **Tests that verify directory structure** - these are done in RobotFramework as part of the scrum workflow (when you add a file you check that it was added by writing a keyword that explains why it was added).
- **Comprehensive testing infrastructure** - with renode for simulation of user scenarios and unit/integration tests for verifying logical and structural integrity.
- **Automatic pipeline that uses your local machine as docker runner** - so that when you push to your repository your pipeline runs on your build machine (or your work machine). (get the full docker image script here)
- **Beautiful documentation** - that is included as a PDF into every release archive.
- **A working merge request workflow** - where nothing is allowed to be committed unless all checks pass and reviews have been completed.

Summary

In this module you learned how to:

- **Add versioning to your apps and releases**
- **Generating software BoM** - so that every file that goes into a release is fully traceable.
- **Generate a release archive** - that also supports building releases for different board versions.
- **Tracing files from BoM to git** - so that you can build on this to implement traceability.

Now your repository is fully prepared for adding all the functional code to it.

The repository with full results and code for this training is available: https://github.com/swedishembedded/sdk

What to do next

- Book call: https://swedishembedded.com/book-call

If you need help getting your own repository in order then Swedish Embedded Group would be happy to help.

You can:

- **Draw on our long experience working with Linux and C** - so that even if you don't have the right skills on your team you will not be stuck.
- **Get your team trained** - so that you can eliminate bottlenecks in your team that are there due to insufficient knowledge about the technology.
- **Get proof of concept done** - so that your team can get the insight necessary in order to quickly put together the full product.

To get all of these benefits and more, book a call with us by going to the calendar page and picking a suitable date and time that works for you.

Click here to go to calendar page and pick a suitable date and time

```
Author: Martin Schröder, 22 Jun 2022
Email: martin.schroder@swedishembedded.com
Training: Zephyr Getting Started Training
Module: 6
```