Ceedling: Basic Porting Guide

At its core, Ceedling is a test build manager. It was originally built to orchestrate the symphony of compilers, linkers, and simulators and blend them with Unity and CMock. The goal was to "Just Work". That's a lofty goal when the subject involves tools from dozens of different vendors, all of which behave differently. Clearly ***some*** configuration was inevitable. Ceedling tries to minimize this by providing smart defaults, and then allowing you, the developer and toolsmith, to tweak nearly every aspect of the process. It's opinionated software, in that the path of least resistance is to follow the useful conventions and paradigms designed into it… but it still provides the hooks to break those conventions where necessary.

This is your guide for doing just that.

Simulator vs Native vs Target

There are three options when it comes to building your unit tests. Two of them are good options. The third is the answer many think is the obvious solution. Most of the time the third option is wrong. Let's take a look at all three.

OPTION 1: SIMULATORS

Many embedded targets have simulators. Often their usefulness is overlooked because they usually do a great job of simulating the processor's instruction set, but a poor job of treating registers, peripherals, and memory the way your target does... instead, simulators often treat all memory locations as if they were RAM.

For Unit Testing, this is AWESOME! Seriously.

Let's say we have a register `USART1_STATUS` and it has a bit `USART_STATUS_OVERFLOW` which is supposed to tell our application that we've overflowed the peripheral's input buffer. If the simulator treats `USART1_STATUS` as RAM, our test can write to `USART1_STATUS` anytime (Even if it's read-only on our real target). Our test can fake a `USART_STATUS_OVERFLOW` and then check to make sure our function handles it correctly.

No muss, no fuss.

To make the simulator option work, it is necessary that a simulator exists for your target (*no, really?*). Also, it is necessary that the simulator allow you to load and run applications from a scriptable interface (like the command line) and that it have some method of collecting results (most often this takes the form of special character write routines that log characters to a file or dump them over standard out).

Positives:

- Tests are built using the same compiler as your release code
- You have full test control over all registers and peripherals

Negatives:

- Simulators can be slow to execute tests
- Configuration can sometimes be complicated (depending on your simulator)

OPTION 2: NATIVE EXECUTABLE

The second good option is to build your tests as native executables. This means that if you are running Microsoft Windows, you build your tests to be run directly on Windows (using Microsoft Visual Studio or MinGW or something like that). If you're running on a Apple OS X or Linux, you're likely going to build using gcc or clang.

There's a trick to this that requires some up-front work: preparing to test registers. Remember how we talked about how awesome it is that we can read or write any register we want on a Simulator? We want that same ability for our native tests! But how can we? Just because address `0x5000C000` is the start of a `GPIO` configuration register set on our target, doesn't mean we can just decide to write to that location on our native host, right? It could be ANYTHING there!

The answer is that we must define our registers in a particular way. We'll get into the details of this on another page, but basically the trick is to take normal register definitions like these:

```
#define PORTG_CONFIG (*(volatile unsigned int*)(0x5000C000))
#define PORTA_CONFIG (*(struct PORT_CONFIG_TYPE*)(0x50000000))
```

During a test, you want them to act more like this:

```
volatile unsigned int PORTG_CONFIG;
volatile struct PORT_CONFIG_TYPE PORTA_CONFIG;
```

Now, if you look closely, the first two instances are common ways of mapping integers and structs to become nice C-friendly registers. The latter two provide the very same interface, but to an integer or struct sitting in RAM. So, if we can make sure we use the first set for our release build and the second for our tests (which is easy, we assure you), then you are ready for testing! (well, once you update all your registers to be this style.)

This topic is discussed further in the How To Test document.

Positives:

- You have full test control over all registers and peripherals
- (Usually) faster builds
- (Usually) (MUCH) faster test execution
- Promotes portable C

Negatives:

- Requires two toolchains (one for release, one for test)
- Might require some small modifications if your compiler has non-standard features
- Upfront time investment for creating flexible register set

OPTION 3: ON TARGET

(PROBABLY THE WRONG CHOICE)

If you've read the other two options, you might be able to guess why executing your tests on the actual embedded target is the worst option. If you guessed "Because you don't have control over the registers" you WIN!

Unit Testing is about testing that your code handles all the situations it might encounter: good and bad. That means you need to be able to inject any error code that you might encounter. It means you will want to write to read-only registers and read from write-only registers. You need access. Executing on the Target doesn't get you that.

That isn't to say there isn't a place for executing code on your target. Quite the contrary, we at ThrowTheSwitch.org are big believers in the power of System Testing. But that's a different animal. For a System Test, you want to treat your final system as a black box. You want to control the external controls and stimuli to your system and measure the responses. It happens against your RELEASE build (please don't use your debug or test builds for this).

Having said all that, there may be the rare case where executing on the target is the best choice (often because it is the only choice, or because you are running multiple styles of tests and have low-level tests written in another way already). If you think this case applies to you, here's our recommendation:

Focus on our helpful hints about using a simulator. When we talk about loading and executing tests on the simulator, replace that with loading and executing tests on your target. When we talk about collecting results from your simulator, replace that with a communication channel through your debugger or usart or similar to collect results from your target.

It CAN be made to work... but there is always some fun configuration to make it happen.

Let's Make A Project

Our first step, no matter which method we've chosen, is to create a project. Open a command line and move to a folder to hold your project. By this we mean a folder which might contain multiple projects. For example, let's say I had a directory structure like this:

- `MyUserFolder`
  - `Projects`
    - `APreviousProject`
    - `AnotherPreviousProject`
    - `ThatProjectThatAlmostMadeItBig`
    - `MyNewProject`

The folder `MyNewProject` doesn't exist yet. It's waiting for us to create it! So we'd open a command prompt at `MyUserFolder/Projects`. From there, we're ready to ask Ceedling to do some work for us:

```
ceedling new MyNewProject --as_gem
```

This will create the folder `MyUserFolder` and populate it with some basic assumptions about how a Ceedling project should be built. We'll be free to change any of those.

One of the things it created for us is a project.yml file. This file is where the majority of our configuration is going to take place. Open it for a moment. At the moment, it doesn't contain a lot. It has a few assumptions. A few prompts for possible modification… but otherwise isn't doing a lot yet. This is because Ceedling is configured to use defaults for almost everything it needs, unless you specify an alternative. This is helpful for getting up and running quickly.

Your first step is to double check the following settings and make changes if you so desire.

```
:project:
  :build_root: build
```

The build root is where Ceedling will be putting all the temporary files during testing. This includes test results, test executables, object files, dependency files, generated C files for runners and mocks, and more. By default, they're all going to get placed in a `build` folder contained in the sample folder which has Ceedling's `project.yml` file.

```
:paths:
  :test:
    - +:/test/**
    - -:/test/support
  :source:
    - src/**
  :support:
    - test/support
```

You'll want to double-check the paths next. Ceedling creates a project with the assumption that you are going to put all your source in a folder named src (possibly organized into subdirectories), all your tests in a folder named test, and that you might have some test support files in a `test/support` folder.

This is a fine assumption, but there are a lot of reasons to switch from it. Maybe your company standardizes on a different structure. Maybe you like to separate your header files. Maybe you have code reused from elsewhere being pulled in. Maybe you just don't like our organization system. In any case, you can add or remove to any of the paths above.

In addition to those shown above, there are also `:include` and `:assembly` paths which can be added to your list. Any of the paths can contain as many or as few lines beneath as you require.

These paths are specified with a little magic. You can see a few of these being used in the default settings above. All are available for you to use:

`+:`	Use this prefix to indicate you are adding a collection of files to a path. This is not explicitly necessary, but makes a logical contrast to the next.
`-:`	Use this prefix to indicate that you will be subtracting any files matching this pattern from the list. Our default setup uses this to avoiding searching the test support directory for actual tests.
`*`	This specifies that we will include all subdirectories depth of 1 below the parent path, and any files in the parent path itself.
`**`	This specifies that we will include all subdirectories recursively discovered below the parent path (including the parent itself)
`?`	Matches a single alphanumeric character
`[x-y]`	Matches a single alphanumeric character within the range specified
`{x,y}`	Matches a single alphanumeric character included in the specified list.

If you need more precision than this, you can also specify individual files to add or subtract to any of these same lists. This is done like this:

```
:files:
  :test:
    - -:/test/test_not_supported.c
  :source:
    - +:/callbacks/comm.c
    - +:/callbacks/comm_*.c
    - -:/callbacks/comm_ethernet.c
    - -:/src/board/atm123.c
```

Some of these patterns can be a little tricky. Luckily, Ceedling has a handy way to let you know what it understood. If you execute any of these commands, Ceedling will output a list of ALL the matches it has made following your criteria. You might have to throw a temporary file into some folders to work it all out, but you'll be glad you did:

```
ceedling files:assembly
ceedling files:header
ceedling files:source
ceedling files:test
```

Setting up the Toolchain

The next step is to set up our toolchain. If you've chosen to go the native route, we have some good news for you: It might be possible to skip this step entirely. By default, Ceedling will assume that gcc is running somewhere on your system. This is going to be true if you're running Linux or macOS (ok, the latter actually runs Clang, but it answers to gcc, so you're fine). On Windows, many developers will have Cygwin or MinGW installed, which will also give them a healthy dose of gcc magic.

But let's say you NEED to make changes. It might be that you just don't like the default options. It might be you don't have your toolchain in your search path. It might be you've chosen to run with one of the other execution methods.

In any case, we're going to be adding a :tools section to our project.yml file. This section allows us to specify the details of our toolchain. At a minimum, we're going to want to specify a compiler and linker. It we're targeting a simulator, we're also going to want to specify a fixture, which is basically our command line calls to run a file on the simulator.

```
:tools:
  :test_compiler:
  :test_linker:
  :test_fixture:
  :test_includes_preprocessor:
  :test_file_preprocessor:
  :test_dependencies_generator:
```

Each of these accepts a number of details. The executable and arguments fields are required. The rest are “optional”, but we really mean that they only need to be included if your toolchain requires that they are included.

```
:tools:
  :test_compiler:
    :executable: acme_cc
    :name: 'acme test compiler'
    :arguments:
      - -I"$": COLLECTION_PATHS_TEST_TOOLCHAIN_INCLUDE
      - -I"$": COLLECTION_PATHS_TEST_SUPPORT_SOURCE_INCLUDE_VENDOR
      - -D$: COLLECTION_DEFINES_TEST_AND_VENDOR
      - --network-license
      - -optimize-level 4
      - "#{`args.exe -m acme.prj`}"
      - -c ${1}
      - -o ${2}
  :test_fixture:
    :executable: tools/bin/acme_simulator.exe
    :name: 'acme test fixture'
    :stderr_redirect: :win
    :background_exec: :none
    :arguments:
      - -mem large
      - -f "${1}"
```

In the example above, our `:test_fixture :executable` is a relative path. Ceedling also supports an absolute path. It also supports specifying just the name of the executable, as you can see in the `:test_compiler`. This assumes that it’s in your path somewhere (see the `:environment` settings if you want to add it to your path).

The `:name` is used strictly in live reporting. You can name the fixture whatever you like.

The `:arguments` list is obviously where you are going to pile all your command line arguments. This is where you should be aware of some opportunities for magic.

`${1}`	This will contain the input or inputs to this step. For example, a compile step will replace ${1} with the name of the C file. A linker step will replace it with a space-delimited list of all object files.
`${2}`	This will contain the output filename of this step.
`#{`cmd`}`	This will execute a shell command BEFORE running this tool and put the results of that command here.
`"$": or $:`	This expects to be followed by one of Ceedling's handy collections. This will create an instance of this string for EACH item in the collection.

The last one is particularly interesting. Ceedling generates internal collections of files constantly. We can take advantage of these collections when executing our own tool calls. For example, the line `-I"$": COLLECTION_PATHS_TEST_TOOLCHAIN_INCLUDE` might become this:

```
-I"src" -I"src/includes" -I"src/stuff" -I"lib/includes"
```

The collections available are these:

- COLLECTION_PATHS_TEST
- COLLECTION_PATHS_SOURCE
- COLLECTION_PATHS_INCLUDE
- COLLECTION_PATHS_SUPPORT
- COLLECTION_PATHS_SOURCE_AND_INCLUDE
- COLLECTION_PATHS_SOURCE_INCLUDE_VENDOR
- COLLECTION_PATHS_TEST_TOOLCHAIN_INCLUDE
- COLLECTION_PATHS_TEST_SUPPORT_SOURCE_INCLUDE
- COLLECTION_PATHS_TEST_SUPPORT_SOURCE_INCLUDE_VENDOR
- COLLECTION_PATHS_RELEASE_TOOLCHAIN_INCLUDE
- COLLECTION_DEFINES_TEST_AND_VENDOR
- COLLECTION_DEFINES_RELEASE_AND_VENDOR

Finally, the `:stderr_redirect` and `:background_exec` are rarely used. When they *are*, it's usually by the `:test_fixture`. The `:stderr_redirect controls` how `$stderr` messages are captured by Ceedling in order to be properly handled. The `:background_exec` allows Ceedling to understand that it shouldn't block on the `:test_fixture` directly. Instead, it should launch it as a background task and wait for the output to know when it's finished.

Compilation Details

You could get away with specifying all your command line arguments in the tools individually instead of using the collections listed above… but Ceedling prefers to separate defines and flags from the tool definition. Not only does this allow us to add defines or flags without changing the tools themselves, it makes it slightly easier for a new project maintainer to find the information. You'll find that this will help immensely if you ever need to port a project to a new toolchain.

Defines

Defines are added to the collection and then injected into your tool as described in the last section. This is particularly useful for specifying Unity & CMock's configuration. These defines are specified as a list, like so:

```
:defines:
  :test:
    - TEST
    - PROCESSOR_LPC1768
    - UNITY_INT_WIDTH=16
    - UNITY_EXCLUDE_FLOAT
  :test_preprocess:
    - PROCESSOR_LPC1768
```

Flags

Similarly, you can specify the flags outside of the tool definition. This might at first seem fairly useless, as you could have just put all of these settings in the tool definition, right? Well, almost. The extra magic here is that you can specify file patterns to pass certain files additional compilation flags, etc. For example:

```
:flags:
  :test:
    :compile:
      :main:
        - -Wall
      :'test_.+':
        - --pedantic
      :*:
        - -foo
```

In this example, if the file matches main, it's going to add the flag `-Wall`. If the file matches `test_` and any other characters, it will add a pedantic requirement. The asterisk specifies that any OTHER files will get the `-foo` flag added to them.

Environment

So, we've fully specified our tools… but sometimes those tools (particularly if you're on a unix-like system) might depend on particular environment variables to be configured. These might be paths or other symbols. Luckily, it's easy for Ceedling to specify a set of environment variables before any of the tools are executed.

```
:environment:
  - :license_server: gizmo.intranet
  - :license: "#{`license.exe`}"
  - :path:
     - Tools/gizmo/bin
     - "#{ENV['PATH']}"
  - :logfile: system/logs/thingamabob.log
```

In our example above, the first represents the most common method of specifying an environment variable: it's a simple key-value pair that will be defined using your platform's symbol creation method.

The next is similar. It allows us to run an executable, and inject the response into the value of a key-value pair. This isn't something that needs to happen often, but you'll be glad you have this option if you DO need it.

If you want to specify the path, like many of the tools we've seen, we can give it a list. Somewhere in that list, we will probably have a line like the one in the second place above. That line pulls in the existing path. Keep that around. You'll want it!

Much like our paths, which had options for dynamic content, there is a Ceedling command for gathering information on how these things are actually working (without running them):

```
ceedling environment
```

That should help debug any issues that might arise.

Customizing CMock and Unity for your Target

Both Unity and CMock make every effort to guess at the optimal settings for your target. However, they're not psychic (maybe that will be in version 3). You will want to refer to the Unity

and CMock configuration documentation for configuring these things. The Unity configuration will take the form of setting defines (see above). The CMock configuration may also include defines, but will also accept a :cmock section in your yaml file. This has been described outside this document.

Fixturing for Simulators and Targets

There is additional fixturing work often required when using a simulator and target. While a native executable is as simple as calling a test and then collecting the results, these other situations may require us to launch tools in the background, trigger the actual tests, detect when the tests have completed, collect the results from somewhere, and possibly do some cleanup before the next round.

We start with the test fixture in the `project.yml` file. Let's take qemu, a popular simulator which can be used to target ARM processors, among many other options. It might look something like this:

```
:tools:
  :test_fixture:
    :executable: qemu-system-arm
    :name: 'qemu'
    :stderr_redirect: :unix
    :arguments:
      - '-cpu cortex-m3'
      - '-M lm3s6965evb'
      - -no-reboot
      - -nographic
      - -kernel ${1}
```

Notice that the options we've selected include specifying the microprocessor to simulate, as well as specifying that we don't want to include graphic support… we are, after all, just running this from the command prompt from within Ceedling. The no-reboot flag specifies that when the application ends, it should close the simulator. This keeps us from having tests that reboot themselves and run forever. Finally, the `${1}` portion injects the actual test we want to run into this command line call.

This takes care of launching our application on an ARM simulator, but it's not good enough. If this were the only fixturing we did, the application would likely fail to start properly within the simulator, and it wouldn't return any of our actual test results. We need to fix both of these things.

First, we add a startup file which can be linked to during the tests. This startup file very likely doesn't need to be nearly as fancy as a release startup file… but it DOES need to do the basics:

Define a vector table, init the variable ram, and then call main. If you're unfamiliar with doing these sort of things, you can probably look at the startup file being used by your project for inspiration.

Once written, we need to tell Ceedling to actually USE this file with all of our tests. This can be done with the :files section. Let's do that… we're also going to add a file for handling our uart, which we will talk about next:

```
:files:
  :support:
    - +:test/support/startup.c
    - +:test/support/uart.c
```

Okay, so our test can now fully compile in a way that the simulator understands (or our target, if we were using real hardware). But we need a way of actually collecting our results. Some simulators will redirect stdout for you. QEMU listens to the simulated target's primary UART and forwards any characters it receives to the environment's stdout. So, if we wish to hear was Unity has to report, we need to use that register. That means our uart.c contains something like this:

```
volatile unsigned int * const UART0DR = (unsigned int *)0x4000c000;

// Qemu's simple model of a UART includes no buffers, control
registers, etc.
void print_uart0(const char s)
{
  *UART0DR = (unsigned int)s; /* Transmit char */
}
```

Pretty eh? We're not done, yet. We also tell Unity that we want it to automatically include a unity_config.h header file and use it to tell Unity to use this new function:

```
#define UNITY_OUTPUT_CHAR(a) print_uart0(a)
```

This covers the two biggest hurdles when switching to using a simulator or the target. Very likely, there are going to be additional details. You'll end up needing to specify a linker file. For a simulator, there's no need to match the real memory footprints… in fact, you can often test on a larger member of the processor family, just to give you more room for mocks and such. For a target build, you can probably use the same linker file you're already using for the release.

Finally, a target build has another fun complication: Actually programming the hardware or collecting the returned data from the UART / USB / whatever your connection is. In these cases, you will often end up writing a batch file to do the majority of the work for you.

Dependency Tracking

When `make` (or other build tools) tackle a release build, the dependencies are almost (but sometimes not quite) obvious. We can follow the includes trail and determine a fairly straightforward tree of dependencies.

This gets a bit more challenging with unit testing. Test runners are automatically generated. Mocks are created from header files, which adds non-header dependencies to test executables that would not otherwise be there. Ceedling does most of this for us. To do this, it juggles a full dependency tree for each test executable. In order to keep the linking time down, it mostly focuses on shallow dependency tracking… by this, it takes a shortcut. Headers that are included by the C files directly are considered dependencies. If there are headers that are included by THOSE headers and so on, these are often ignored. This will mean that if you change a deep dependency, Ceedling isn't going to notice when running by defaults. This is a good trade for many users. If things act weird, they issue a clobber first. Their continuous integration servers ALWAYS issue a clobber, to make sure they don't run into this issue.

Depending on the complexity of your system, this might be acceptable. If there are a lot of deep dependencies in your system, though, then you're going to want to enable the following setting:

```
:use_deep_dependencies: TRUE
```

With this enabled, Ceedling will always track your dependencies as deeply as it is able. This will add significant time to your test builds, but not as much time as clobbering with every test.

Giving Yourself Options

The configuration we've covered thus far has focused on projects where the modules all of a specific single configuration. This is will most often be true, but, as many embedded developers have experienced, there are situations where a single codebase can be used to produce multiple releases. These situations are often driven by including / excluding certain files and through combinations of defines.

This is where ceedling option files come in. You start by specifying different defines, flags, and paths, as required by a configuration, into a new yaml file. Name this yaml file something descriptive and place it in a folder (like `options/advanced.yml` for example).

Then when you call ceedling, the first part of the call should specify which set of options you plan to use. This is named the same thing as the yaml file (without the path or extension), like so:

```
ceedling options:* test:all
```
You can configure the path that it will look for options files by setting it here:

```
:project:
  :options_path: options
```

This will handle most situations to run into. There's an even stronger method of separating out your options, though. What if you needed to make MAJOR changes, like one version of your codebase is meant to be compiled with one compiler, while another version is compiled using a completely different toolchain.

In this case, you might want to step up to specifying multiple projects. In this case, each project is a complete yaml file, as we've been specifying, but named something other than project.yml. When we call ceedling and want to use one of these, we start by specifying the project we wish to use:

```
ceedling project:cortex_m7 test:all
```

We've worked with people that have used this functionality to provide multiple testing methods. For example, they might have a project for testing in a simulator and another for native tests. These might be used by different users or the CI server. We're not going to lie, doing both WILL mean more work. We wouldn't go so far as to say we recommend this approach, but we will say we've seen it used effectively!

Debugging Your Configuration

There is a LOT of configuration possible with Ceedling… and honestly we've just hit the highlights in this document. There are many more things you can do. Check out the documents on Releases, Plugins, and so on. Also, you'll want to look at the main project documentation for Ceedling if you really need to dig in.

Clearly, with that much configuration, there is plenty of potential for things to go wrong. In addition to the information-gathering methods we've already mentioned, be aware that you can also specify the following options.

```
ceedling logging
ceedling verbosity[x]
```

These are meant to be chained, much like the way project or option settings were. The verbosity setting accepts a number between 0 and 4, basically following this table:

- 0 - Silent
- 1 - Errors Only

- 2 - Complain (Errors, Warnings, and Similar Notices)
- 3 - Normal (Errors, Warnings, Notices, and Status Messages)
- 4 - Obnoxious (All Normal plus Extra Verbose Verification Messages)
- 5 - Debug (For Hardcore Debugging)

For example, you could execute the following:

```
ceedling verbosity[5] clobber test:all
```

And you will have plenty of reading material to sift through.

Final Words

We hope you've found this porting guide helpful. Keep in mind that there are more details in the CeedlingPacket.md that comes with the project, as well as additional documents packaged with this class. If none of these are sufficient, you are always welcome to reach out on the throwtheswitch.org/forums.

Happy Testing!