How to Use Ceedling for Embedded Test-Driven Development

with Step-by-Step Examples

Matt Chernosky

http://electronvector.com

Contents

Welcome

Maybe you've heard of **Test-Driven Development** (TDD), and maybe you've even thought it seemed like a reasonable idea. **If you haven't tried TDD yet though, you really should.**

This guide contains step-by-step examples to get you started test driving in C, especially for embedded software applications.

We'll look at how to use the unit test framework called Ceedling to help us do this. In the first example, we'll see how to create tests and write the code to make them pass. In the second example we look at mocking, and learn how to use it simulate our hardware.

All the tests in these examples compile and run on your host PC (with GCC), with no target hardware needed.

What is Test-Driven Development (TDD)?

The premise of TDD is that we use the creation of unit tests to incrementally drive the development of the software. The steps look like:

1. Write a test, and watch it fail.
2. Implement just enough code to make the test pass.
3. Refactor.
4. Repeat.

This allows us to be very clear about what the code is to do, because we've defined every behavior in a test.

With each iteration you add a bit more functionality to your software, and the tests give you confidence that you're doing it correctly.

One of the difficulties of unit testing is that it takes some degree of experience to write testable code. If you write the tests first though, you'll figure out how to make your code testable while you write it. This means you won't waste your time with this problem at all!

For me, the greatest aspect of TDD is that it takes a big problem to solve (how to implement a software application) and reduces it to a simple problem (what is the next little thing I need this software to do). Then I just write a test for that feature, implement it and repeat.

You simply stop when you don't need any more features. And when you do stop, you're confident that what you have is working exactly how you want it to. It's unlikely that you'll be spending a lot of time chasing down bugs.

The Tools

The testing tools used in this example are Ceedling, Unity and CMock. These are the best C unit test tools available, from the people over at http://www.throwtheswitch.org/.

Ceedling is an automated testing framework for C applications.

In order to do TDD, you need to be able to create and run tests easily since you'll be doing it *all the time*. Ceedling provides automatic test discovery, mock generation and test execution, which makes it the best option for unit testing in C. It also builds and runs tests on the host PC, so when working on an embedded project we don't have to waste time downloading to the target.

Unity is the unit test framework provided with Ceedling. It gives us all of our test assertions constructing our tests.

CMock is the mocking framework used with Ceedling. The mocking framework is what lets us simulate interactions with other software modules, so that we can test our software units in isolation.

Installing Ceedling

Ceedling requires Ruby to run and uses GCC to build each test.

1. Install Ruby

   Windows installer: https://dl.bintray.com/oneclick/rubyinstaller/rubyinstaller-2.3.1.exe
   Other instructions: https://www.ruby-lang.org/en/documentation/installation/.

   Be sure that the Ruby bin folder is in your path, e.g. C:\Ruby23\bin

2. Install Ceedling with the Ruby "gem" tool with the command: `gem install ceedling`

   Installation Error: "certificate verify failed"

   I've recently had an issue trying to use the "gem" command. It's a problem with the RubyGems server (where Ceedling and other gems are hosted). If you get an error like this:

   ```
   ERROR:  Could not find a valid gem 'ceedling' (>= 0),
   here is why:
         Unable to download data from https://rubygems.org/ -
   SSL_connect returned=1 errno=0 state=SSLv3 read server
   certificate B: certificate verify failed (
   https://api.rubygems.org/specs.4.8.gz)
   ```

   The quick fix is to install a new certificate into Ruby. Download this certificate file:

   https://raw.githubusercontent.com/rubygems/rubygems/master/lib/rubygems/ssl_certs/index.rubygems.org/GlobalSignRootCA.pem

   *and place it into **C:\Ruby23\lib\ruby\2.3.0\rubygems\ssl_certs**.*

   Note that your path may be different if you installed to another folder or installed a different version.

3. If you're on Windows, you'll likely need to install GCC.

   I recommend installing with Cygwin (https://cygwin.com/install.html).

   When installing be sure to select the "Devel" packages to have GCC installed. Then put the Cygwin bin folder in your path, e.g. C:\cygwin64\bin.

Creating a New Ceedling Project

Use the `ceedling new <projectName>` command to create a new project:

```
$ ceedling new MyProject
create  MyProject/vendor/ceedling/docs/CeedlingPacket.pdf
create  MyProject/vendor/ceedling/docs/CExceptionSummary.pdf
...
create  MyProject/vendor/ceedling/vendor/unity/src/unity.h
create  MyProject/vendor/ceedling/vendor/unity/src/unity_internals.h
create  MyProject/project.yml

Project 'MyProject' created!
- Tool documentation is located in vendor/ceedling/docs
- Execute 'ceedling help' to view available test & build tasks
```

This generates a project tree and the configuration files needed to use Ceedling. Project creation only needs to be done once when starting a project. Important among the created folders are:

- **src**: Where all of our source files will go.
- **build**: Contains anything generated by Ceedling during the build.
- **test**: Where our unit test files will go.

Now we have a project in the **MyProject** folder. Note the instructions from the Ceedling output when we created the project -- we can use `ceedling help` to show us how to use it:

```
$ ceedling help
ceedling clean                          # Delete all build artifacts and
                                        temporary products
ceedling clobber                        # Delete all generated files (and
                                        build artifacts)
ceedling environment                    # List all configured environment
                                        variables
ceedling files:header                   # List all collected header files
ceedling files:source                   # List all collected source files
ceedling files:test                     # List all collected test files
ceedling logging                        # Enable logging
ceedling module:create[module_path]     # Generate module (source, header and
                                        test files)
ceedling module:destroy[module_path]    # Destroy module (source, header and
                                        test files)
```

```
ceedling paths:source                # List all collected source paths
ceedling paths:support               # List all collected support paths
ceedling paths:test                  # List all collected test paths
ceedling summary                     # Execute plugin result summaries (no
                                     build triggering)
ceedling test:*                      # Run single test ([*] real test or
                                     source file name, no path)
ceedling test:all                    # Run all unit tests (also just
                                     'test' works)
ceedling test:delta                  # Run tests for changed files
ceedling test:path[dir]              # Run tests whose test path contains
                                     [dir] or [dir] substring
ceedling test:pattern[regex]         # Run tests by matching regular
                                     expression pattern
ceedling verbosity[level]            # Set verbose output (silent:[0] -
                                     obnoxious:[4])
ceedling version                     # Display build environment version
                                     info
```

Example #1: Starting TDD with Ceedling

In this example we'll walk through a single TDD micro-cycle, adding a feature by writing a test and getting it to pass.

Note that a new Ceedling project must have been created as described in the previous section.

Create a Module

Now it's time to write some code. Imagine we're building a car and we want to build a module to implement the lighting system. We create a module like this:

```
$ ceedling module:create[lights]
Generating 'lights'...
mkdir -p ./test/.
mkdir -p ./src/.
File ./test/./test_lights.c created
File ./src/./lights.c created
File ./src/./lights.h created
```

This creates three files: **lights.c** to implement our module, **lights.h** to define the public interface and a test file where we can put the unit tests for it. These files are automatically created in the correct folders of our tree.

Note: We could also have provided a deeper path in which to create the module, e.g. `rake module:create[electrical/lights]`.

At this point, we can try running our unit tests with the `ceedling` command:

```
$ ceedling

Test 'test_lights.c'
--------------------
Generating runner for test_lights.c...
Compiling test_lights_runner.c...
Compiling test_lights.c...
Compiling unity.c...
Compiling lights.c...
Compiling cmock.c...
Linking test_lights.out...
Running test_lights.out...
```

```
-----------
TEST OUTPUT
-----------
[test_lights.c]
  * ""

--------------------
IGNORED TEST SUMMARY
--------------------
[test_lights.c]
  Test: test_module_generator_needs_to_be_implemented
  At line (14): "Implement me!"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  0
FAILED:  0
IGNORED: 1
```

This tells us that a single test was run and it was *ignored*.

The `module:create` operation has used a template to create the test file. Inside the test file **test_lights.c** is a single test named `test_module_generator_needs_to_be_implemented` which uses a special *ignore* directive to tell Ceedling to ignore this test. The function looks like this:

```
void test_module_generator_needs_to_be_implemented(void)
{
  TEST_IGNORE_MESSAGE("Implement me!");
}
```

This is the convention for unit tests which Ceedling. Test files have names that start with `test_` and they go in the **test** folder. Within each of these files, unit tests are functions whose names start with `test_`.

Also in the test file are the `setUp()` and `tearDown()` functions. These functions are run before and after each of the test functions in the test file. These functions are yours to use if you need them.

Implement a Feature

Now that we have a module for the lights, it's time to add some functionality. In the test-driven way, we'll first add a test that describes some desired behavior. Say we want this behavior:

When the headlight switch is off, then the headlights are off.

In this case we're going to replace `test_module_generator_needs_to_be_implemented()` with a new test function:

```
void test_WhenTheHeadlightSwitchIsOff_ThenTheHeadLightsAreOff(void)
{
  // When the headlight switch is off...
  lights_SetHeadlightSwitchOff();

  // then the headlights are off.
  TEST_ASSERT_EQUAL(false, lights_AreHeadlightsOn());
}
```

What we've done here is define two new functions to implement in the **lights** module, `lights_SetHeadlightSwitchOff()` and `lights_AreHeadlightsOn()`. We call the first function to turn the lights off, and then call the second to confirm the state of the headlights.

The `TEST_ASSERT_EQUAL()` macro is what we use to verify that the value returned from `lights_AreHeadlightsOn()` is the expected value (`false`). This is one of the many macros available for comparing various types, all of which are explained in the Unity documentation (https://github.com/ThrowTheSwitch/Unity/tree/master/docs).

Now we can run our tests, but obviously this is going to fail with all kinds of compilation errors, because these functions don't even exist yet.

```
$ ceedling

Test 'test_lights.c'
--------------------
Generating runner for test_lights.c...
Compiling test_lights_runner.c...
Compiling test_lights.c...
...
> Shell executed command:
'gcc.exe -I"test" -I"test/support" -I"src"
   -I"MyProject/vendor/ceedling/vendor/unity/src"
```

```
   -I"MyProject/vendor/ceedling/vendor/cmock/src"
   -I"build/test/mocks" -DTEST -DGNU_COMPILER -g
   -c "test/test_lights.c" -o "build/test/out/test_lights.o"'
> And exited with status: [1].

...
ERROR: Ceedling Failed
```

The next step is to implement the minimum functionality to pass our test. Here is the interface defined in **lights.h**:

```
#ifndef lights_H
#define lights_H

#include <stdbool.h>

void lights_SetHeadlightSwitchOff(void);
bool lights_AreHeadlightsOn(void);

#endif // lights_H
```

And the implementation in lights.c:

```
#include "lights.h"
#include <stdbool.h>

void lights_SetHeadlightSwitchOff(void)
{
}

bool lights_AreHeadlightsOn(void)
{
  return false;
}
```

We can then run our test and see that it passes:

```
$ ceedling

Test 'test_lights.c'
```

```
--------------------
Compiling test_lights.c...
Compiling lights.c...
Linking test_lights.out...
Running test_lights.out...

-----------
TEST OUTPUT
-----------
[test_lights.c]
  * ""

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
```

This may sort of feel like we're cheating, since `lights_SetHeadlightSwitchOff()` doesn't actually do anything and `lights_AreHeadlightsOn()` simply returns `false`, but as we add more tests we'll continue to add functionality.

Repeat

We can now continue adding “features” to the module until it does every thing that we need it to -- by adding tests and then writing the code to make them pass. For example we might want to implement this behavior:

When the headlight switch is on, then the headlights are on.

So, we create an additional test:

```
void test_WhenTheHeadlightSwitchIsOn_ThenTheHeadLightsAreOn(void)
{
  // When the headlight switch is on...
  lights_SetHeadlightSwitchOn();

  // then the headlights are on.
  TEST_ASSERT_EQUAL(true, lights_AreHeadlightsOn());
}
```

If we run this test it will fail because `lights_SetHeadlightSwitchOn()` doesn't exist yet, but when we update **lights.h**:

```
#ifndef lights_H
#define lights_H

#include <stdbool.h>

void lights_SetHeadlightSwitchOff(void);
void lights_SetHeadlightSwitchOn(void);
bool lights_AreHeadlightsOn(void);

#endif // lights_H
```

And add the implementation in **lights.c**:

```
#include "lights.h"
#include <stdbool.h>

static bool areLightsOn = false;

void lights_SetHeadlightSwitchOff(void)
{
    areLightsOn = false;
}

void lights_SetHeadlightSwitchOn(void)
{
    areLightsOn = true;
```

```
}

bool lights_AreHeadlightsOn(void)
{
  return areLightsOn;
}
```

Then we can run our tests and watch them both pass:

```
$ ceedling

Test 'test_lights.c'
--------------------
Generating runner for test_lights.c...
Compiling test_lights_runner.c...
Compiling test_lights.c...
Linking test_lights.out...
Running test_lights.out...

-----------
TEST OUTPUT
-----------
[test_lights.c]
 + ""

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  2
PASSED:  2
FAILED:  0
IGNORED: 0
```

Refactoring

Every time we get a test to pass, is an opportunity to refactor. Since we have a suite of passing tests, we can change the code in any way and we'll immediately know if we broke something. This allows us the ability to freely experiment with improving the code, e.g. to make it simpler or easier to understand.

Refactoring can also be done to the tests as well. You may notice that as you add more and more tests, some become redundant or you end up with a lot of duplication across tests. You'll want to keep this under control as you work, so that they tests don't get too difficult to understand.

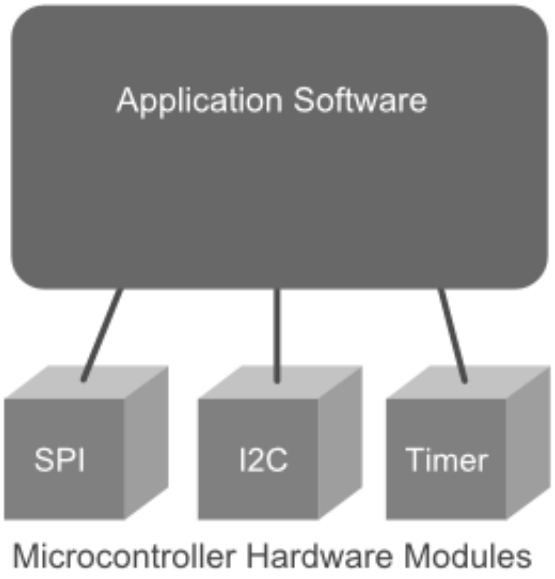
Mocking Hardware Interfaces

How can you unit test your embedded software? What about your hardware dependencies?

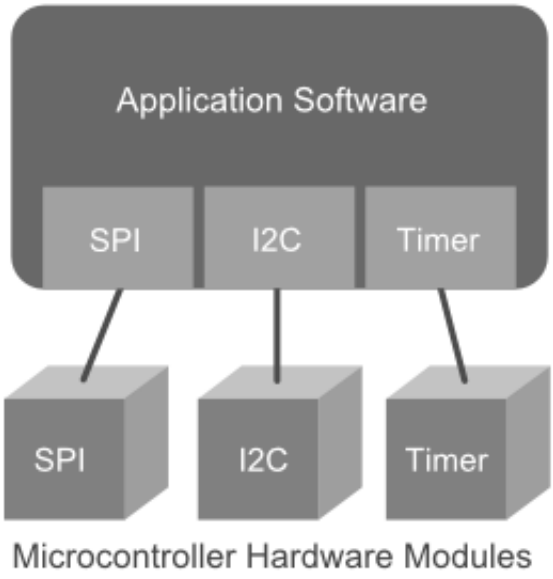
The secret is *mocking*.

We can mock the interfaces to our hardware so that we **don't need the actual hardware to test**. This allows us to run our tests more quickly and before the hardware might even be available.

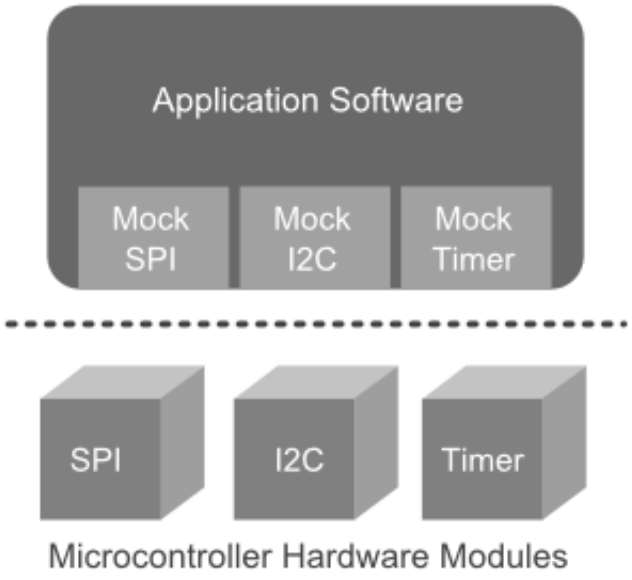
If we're developing the software for an embedded microcontroller, we're probably going to be using the microcontroller-provided hardware modules for things like SPI, I2C, timers, etc.



For each of these hardware interfaces, we want to have a corresponding software module containing the microcontroller hardware dependencies (i.e. hardware register accesses).



We can then mock each of these hardware interfaces, eliminating our hardware dependencies but still allowing us to unit test our application. Instead of compiling these tests for the embedded microcontroller, we compile them for and run them on our host PC.

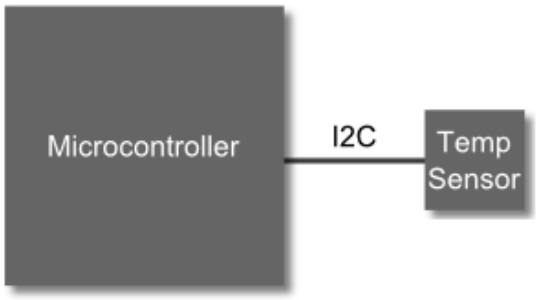


To help you create your mocks you want to use a *mocking framework*. The mocking framework included with Ceedling is CMock. It allows you to create mocks of individual software modules from their header files. **Ceedling improves the experience by automatically using CMock to generating the mocks that we need.**

Example #2: Mocking a Hardware Interface with CMock

Note that this example assumes that we already have an existing Ceedling project. See the section titled Creating a New Ceedling Project if you need help.

Imagine that we want to talk to an external I2C temperature sensor.



Create the Temperature Sensor Module

Let's create a module that will be our temperature sensor driver.

```
$ ceedling module:create[tempSensor]
Generating 'tempSensor'...
mkdir -p ./test/.
mkdir -p ./src/.
File ./test/./test_tempSensor.c created
File ./src/./tempSensor.c created
File ./src/./tempSensor.h created
```

Write Our First Test

What is the first thing we want to be able to do with this sensor? I'd like to be able to read the current temperature value.

Cool. So I take a look at the datasheet for my fictional temperature sensor and I can see that it has a bunch of 16-bit registers -- each with 8-bit addresses -- one of which is the temperature register.

The scaling of the values is such that a register value of 0 is -100.0°C and a register value of 0x3FF is +104.6°C. This makes each bit equivalent to 0.2°C.

Register Value	Temperature
0x000	-100.0 °C
0x1F4	0.0 °C
0x3FF	+104.6 °C

Now lets add our first test to t**est_tempSensor.c**. I want to know that when I read a temperature register value of 0x3FF that the temperature calculated is 104.6.

```
void
test_whenTempRegisterReadsMaxValue_thenTheTempIsTheMaxValue(void)
{
    uint8_t tempRegisterAddress = 0x03;
    float expectedTemperature = 104.6f;
    float tolerance = 0.1f;

    //When
    i2c_readRegister_ExpectAndReturn(tempRegisterAddress, 0x3ff);

    //Then
    float actualTemperature = tempSensor_getTemperature();
    TEST_ASSERT_FLOAT_WITHIN(tolerance, expectedTemperature,
            actualTemperature);
}
```

First we set up some variables to hold our expected values. Then in the "when" clause, we need to simulate (or mock) the I2C module returning a value of 0x3ff on a read of the temperature address.

For the moment, we pretend that there is another i2c module (it doesn't actually exist yet) which handles the I2C communication with the temperature sensor. This is where our hardware dependent code will eventually go.

So, the `i2c_readReadgister_ExpectAndReturn` function is actually a mock function used to simulate a call to a function called `i2c_readRegister` in the i2c module. We'll come back to this in a moment.

The "then" clause is where we test that the tempSensor module actually returns the correct temperature when we call `tempSensor_getTemperature`. This function doesn't exist yet either.

Create the Function Under Test

Lets create the `tempSensor_getTemperature` function with a dummy implementation:

tempSensor.h:

```
    # ifndef tempSensor_H
    # define tempSensor_H

    float tempSensor_getTemperature(void);

    # endif // tempSensor_H
```

tempSensor.c:

```
    # include "tempSensor.h"

    float tempSensor_getTemperature(void)
    {
        return 0.0f;
    }
```

Mock the I2C Interface

If we try and run the test now, the compiler will complain that it doesn't know about the `i2c_readReadgister_ExpectAndReturn` mock function. This is because the `i2c_readRegister` function doesn't exist and we haven't yet told Ceedling to mock it.

We don't actually need to implement this function however. It's enough to declare the function prototype in a header file and tell Ceedling to mock it with CMock.

Create the header file, **i2c.h**:

```
# ifndef i2c_H
# define i2c_H

# include <stdint.h>

uint16_t i2c_readRegister(uint8_t registerAddress);

# endif // i2c_H
```

The way we tell Ceedling to mock this module is to add this line to **test_tempSensor.c**:

```
# include "mock_i2c.h"
```

This tells Ceedling: *You know the i2c.h header you see over there? Well... use CMock to generate the implementation and compile it in for us, okay?*

When CMock gets a hold of the header file it looks at all the functions defined there and generates several mock functions for each... including the `i2c_readRegister_ExpectAndReturn` function we used in the test. This mock function appends an additional argument to the original `i2c_readRegister` function, which is the value we want the function to return to the calling function.

For more details on all the mock functions available with CMock, see the CMock documentation at: https://github.com/ThrowTheSwitch/CMock/blob/master/docs/CMock_Summary.md#generated-mock-module-summary.

Implement the Function Under Test

Now we can implement the logic for our `tempSensor_getTemperature` function. Our new tempSensor.c is:

```
# include "tempSensor.h"
# include "i2c.h"
# include <stdint.h>

float tempSensor_getTemperature(void)
{
    uint16_t rawValue = i2c_readRegister(0x03);

    return -100.0f + (0.2f * (float)rawValue);
}
```

If we run our test, it should pass now.

Adding Another Test

We'll next want to add more tests for other possible return values from `i2c_readRegister`. This is easily done by changing the return value provided to the mock function.

For example, to test that the minimum temperature value is read correctly:

```
void
test_whenTempRegisterReadsMinValue_thenTheTempIsTheMinValue(void)
{
    uint8_t tempRegisterAddress = 0x03;
    float expectedTemperature = -100.0f;
    float tolerance = 0.1f;

    //When
    i2c_readRegister_ExpectAndReturn(tempRegisterAddress, 0x0);

    //Then
    float actualTemperature = tempSensor_getTemperature();
    TEST_ASSERT_FLOAT_WITHIN(tolerance, expectedTemperature,
        actualTemperature);
}
```

Now we have a driver for an external hardware device that we can test without any of the hardware. We can continue to develop the driver — adding more tests and features — by building and testing on our host PC.

By putting all of the microcontroller-dependent I2C operations into their own module, we easily mocked them with Ceedling and CMock. In fact, we didn't even have to implement this module yet — we just had to define its interface in the header file.

Using our mocks, we created unit tests that verify the behavior of our temperature sensor driver. **As the rest of our application is developed, we can easily run these unit tests at any time to make sure the driver will still work correctly.**

Example #3: Add Unit Tests to Your Current Project with Ceedling

You want to try unit testing your embedded software but there's a problem — you've got an existing project and a whole lot of code already written. Maybe it's even embedded legacy code.

You can build, load and run your application just fine from your IDE. But where do the tests go and how do you run them? And what does it mean for your existing project?

Well, it turns out that you can add Ceedling to your project and run it independently from your IDE and release build.

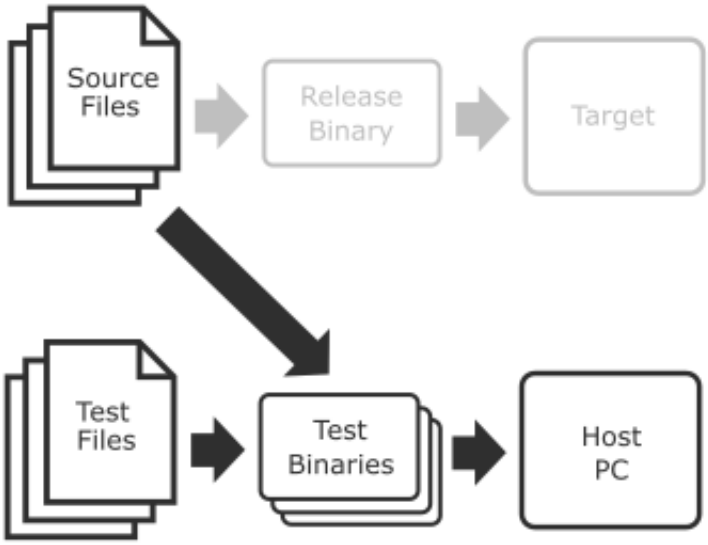
The test code (and framework) will be isolated from your production code and won't interfere with your release builds. This allows you to experiment with unit testing... without messing with the rest of your team.

Using Ceedling like this is the quickest way to get Unity and CMock set up to test your code. Don't worry about integrating with Eclipse (or whatever IDE you're using) yet -- just get your tests running from the command line first.

Typically, you have some source files that your IDE compiles into a release build that can be run on the target:

To add unit testing support to this project you can set up Ceedling to run in parallel from the same source files. You write the tests in separate test files, and then Ceedling uses GCC (instead of your target compiler) to build tests that you run on your host PC.

These test binaries are built in their own build folder so they don't interfere with your existing release configuration. The tests are executed independently from your IDE by running just a few simple commands from the command prompt.

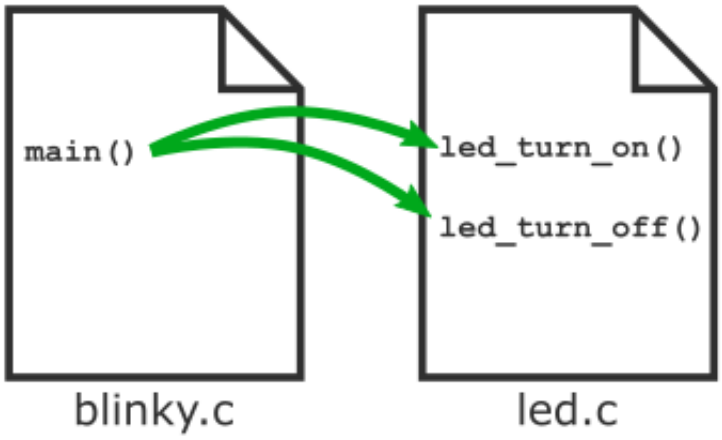


Start with an Existing Project

In this exercise I'm using an example project for the TI Tiva C Series LaunchPad development board. It's a simple little board with an ARM Cortex-M4.

The example is based on TI's "blinky" project, which just blinks an LED. We'll be modifying this code as we progress, but you can find the starting point for this exercise here on GitHub.

The project source consists of a main loop in **blinky.c** and an LED driver in **led.c**. The `main()` loop just calls into the LED driver with `led_turn_on()` and `led_turn_off()` to do the blinking:



This project also has all of the "junk" in it that you'd expect when using an IDE. In fact it has project settings and build folders for Keil, IAR or Code Composer Studio (TI's own free Eclipse-

based IDE). Here's a simplfied view of what this mess looks like (some files are omitted for brevity):

Notice here that the source files are all mixed-in with other types of files here. I am not a fan of this nonsense... but this is pretty common especially with IDEs from embedded vendors. I much prefer the convention of putting all the source in it's own folder. This is cleaner, makes the project tree easier to understand, and also makes it easier to configure Ceedling. We'll revisit this a little later.

For now though — since we're just getting started — we'll leave everything as it is here and install Ceedling along side of it. That way we won't break anything in this existing project.

Install and Configure Ceedling

Before you can add Ceedling to your project you'll need to install Ceedling on your system. This also requires installing Ruby and GCC. See the earlier sections of this guide for help with this.

Once Ruby, GCC and Ceedling are installed, the first step is to install Ceedling into the existing project. This is done from the command line.

WARNING: *This is going to dump a bunch files into your project. As with any project changes make sure you've got a backup somewhere — preferably in source control.*

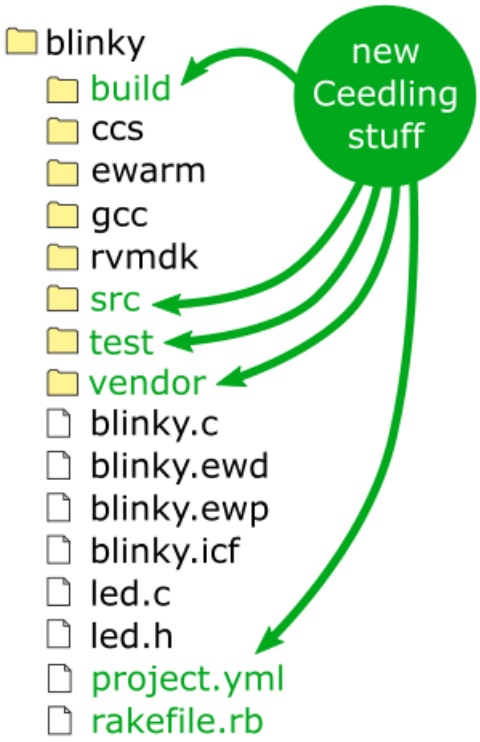
Ceedling has a `new` command for creating "new" projects. It's not obvious, but you can also use `new` to install Ceedling into an existing project. Let's check it out.

So from the command line, go in to the parent folder of your project. In this case, it's the folder above our blinky project. From there you'll run `ceedling new blinky` (since our project is in a folder named "blinky"). This will install Ceedling into your existing project folder by creating some new files and folders:

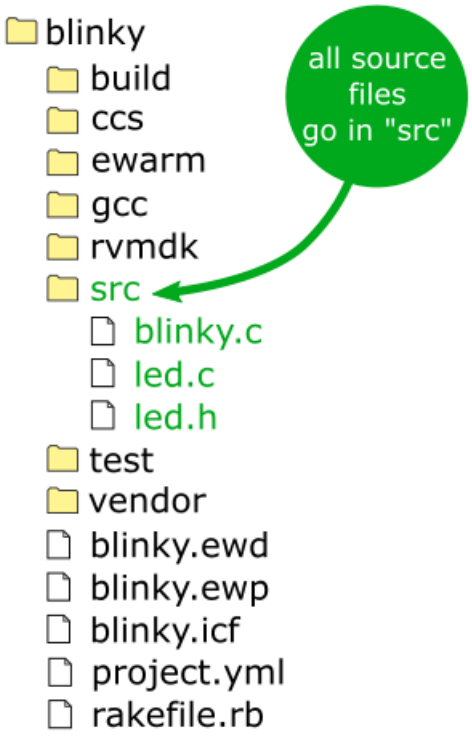
```
projects> ceedling new blinky

Welcome to Ceedling!
      create  blinky/vendor/ceedling/docs/CeedlingPacket.pdf
      create  blinky/vendor/ceedling/docs/CExceptionSummary.pdf
      ...
      create  blinky/vendor/ceedling/vendor/unity/src/unity_inte...
      create  blinky/project.yml

Project 'blinky' created!
- Tool documentation is located in vendor/ceedling/docs
- Execute 'ceedling help' to view available test & build tasks
Now you can drop into the blinky project folder and run Ceedling
with ceedling test:all. We've haven't created any tests yet though,
so no tests are actually going to execute:

projects> cd blinky

projects\blinky> ceedling test:all

---------------------
OVERALL TEST SUMMARY
---------------------

No tests executed.
```

Ceedling just added these files and folders to the project:

- **build**: This is where the tests are built.
- **src**: This is where Ceedling expects to find your source code.
- **test**: This is where your unit tests will go.
- **vendor**: This is where the Ceedling source files are.
- **project.yml**: This is the configuration file for Ceedling.



Notice that Ceedling expects the source code to be in the **src** folder. It's time to move the source code into the **src** folder. For blinky, this means moving **blinky.c**, **led.h** and **led.c**. Unfortunately this might require some changes in your IDE to handle this new folder in the project tree, but this is the best way to set up your project.



Note that you can have any folder tree that you want below **src**, so you can move any existing source folders in there as well.

Ceedling Tip: You can confirm that Ceedling knows about your source files by running `ceedling files:source`:

```
projects\blinky> ceedling files:source
source files:
 - src/blinky.c
 - src/led.c
file count: 2
```

Yeah! Now Ceedling is installed in our project and ready to go.

Create a New Test File

Now that Ceedling is installed, it's time to add some tests. The LED driver (**led.c**) is a good candidate here because it's an isolated module. Before we can add tests we'll need a new test file to put the them in.

Ceedling makes it easy to create the test files for existing modules with its `module:create` command. Typically this command creates a **.c**, **.h** and a test file for a new source module. If a file already exists though, the file is left untouched. This means we can use it to easily create a test file for **led.c**:

```
projects\blinky> ceedling module:create[led]

Generating 'led'...
mkdir -p ./test/.
mkdir -p ./src/.
File ./test/./test_led.c created
File ./src/./led.c already exists!
File ./src/./led.h already exists!
```

The test file it created is **test/test_led.c**. This is built from a template that includes the header files and the `setUp()` and `tearDown()` functions needed by any test. This saves us the time of having to manually copy/paste/edit this from another test file.

Getting it to Build

Now that we have a test file for our LED module, we need to get it to build. Here's where the real fun begins! In this step we're going to chase down a bunch of errors as we try to find the "seams" of the LED module so that we can test it in isolation.

This is going to involve setting up some mocks and configuring Ceedling. We're just going to read the error messages and use them to figure out what needs to be fixed at each stage.

Adding More Source Folders

After adding our first test file **test_led.c**, if we try to run the tests we get our first error:

```
projects\blinky> ceedling test:all

Test 'test_led.c'
-----------------
Generating runner for test_led.c...
Compiling test_led_runner.c...
Compiling test_led.c...
Compiling unity.c...
Compiling led.c...
src/led.c:5:27: fatal error: inc/hw_memmap.h: No such file or
directory
 #include "inc/hw_memmap.h"
                           ^
```

If we take a look at **led.c** it includes a couple files from our processor library: **hw_memmap.h** and **gpio.h**. These are driver files provided by TI for controlling the GPIO:

```
#include "led.h"

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "driverlib/gpio.h"

void led_turn_on(void)
{
  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
}

void led_turn_off(void)
{
  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
}
```

When I installed TivaWare, these files were installed to **C:\ti\TivaWare_C_Series-2.1.2.111** but Ceedling thinks that all of our source files are in the **src** folder. We need to tell Ceedling how to look in this other folder for source as well.

Ceedling is configured in the **project.yml** file. This is a YAML file that Ceedling loads each time it is run. In **project.yml** there is a section for `:paths:` which includes settings for `:test:`, `:source:`, and `:support:`. You add another source path (like **C:\ti\TivaWare_C_Series-2.1.2.111**) by adding another path to the `:source:` list:

```
:paths:
  :test:
    - +:test/**
    - -:test/support
  :source:
    - src/**
    - C:\ti\TivaWare_C_Series-2.1.2.111  # The new source path.
  :support:
    - test/support
```

Mocking Hardware Drivers from the Header Files

With the source path for the TI drivers set, we can try to run Ceedling again:

```
projects\blinky> ceedling test:all

Test 'test_led.c'
-----------------
Generating runner for test_led.c...
Compiling test_led_runner.c...
Compiling test_led.c...
Compiling unity.c...
Compiling led.c...
Compiling cmock.c...
Linking test_led.out...
build/test/out/led.o: In function `led_turn_on':
projects/blinky/src/led.c:10: undefined reference to `GPIOPinWrite'
```

Now that Ceedling can find **driverlib/gpio.h**, it knows that it needs to link in a `GPIOPinWrite` function. Now, we're not going use the real function since we're running on the host PC. So we need to mock it. We can mock all of the functions in **gpio.h** in our test by adding `#include "mock_gpio.h"` to **test_led.c**:

```
#include "unity.h"
#include "led.h"

#include "mock_gpio.h"  // This will mock the functions in
driverlib/gpio.h.

void setUp(void)
{
}

void tearDown(void)
{
}

void test_module_generator_needs_to_be_implemented(void)
{
    TEST_IGNORE_MESSAGE("Implement me!");
}
```

CMock Won't Do Paths to Header Files

Now we're getting somewhere! Let's try running Ceedling again:

```
projects\blinky> ceedling test:all

Test 'test_led.c'
-----------------
ERROR: Found no file 'gpio.h' in search paths.
rake aborted!
```

Oh, so Ceedling can't find **driverlib/gpio.h** so that it can mock it. Remember how we included **mock_gpio.h** in the test? Well Ceedling is looking in all of its configured source folders for **gpio.h**, not **driverlib/gpio.h**. We need add the **driverlib** folder to the source paths so that it can find **gpio.h** in there:

```
:paths:
  :test:
    - +:test/**
    - -:test/support
  :source:
    - src/**
    - C:\ti\TivaWare_C_Series-2.1.2.111
    - C:\ti\TivaWare_C_Series-2.1.2.111\driverlib  # To find gpio.h.
  :support:
    - test/support
```

Including Other Header Files in Our Mocks

What will the next error be?? Running the tests again gives us this one:

```
projects\blinky> ceedling test:all

Test 'test_led.c'
-----------------
Creating mock for gpio...
WARNING: No function prototypes found!
Generating runner for test_led.c...
Compiling test_led_runner.c...
In file included from build/test/mocks/mock_gpio.h:5:0,
                 from build/test/runners/test_led_runner.c:30:
C:/ti/TivaWare_C_Series-2.1.2.111/driverlib/gpio.h:153:50: error:
unknown type name 'bool'
 extern uint32_t GPIOIntStatus(uint32_t ui32Port, bool bMasked);
```

Okay. So this is a problem with using off-the-shelf code from somewhere else (thank you TI). These TivaWare driver files (like **gpio.h**) are set up strangely. Even though **gpio.h** needs **stdbool.h** and **stdint.h** it doesn't actually `#include` them. As the user, you're supposed to include them in your source file before including **gpio.h**.

Unfortunately this means we need to include **stdbool.h** and **stdint.h** in our auto-generated mock files. Fortunately Ceedling has a setting for this in the `:cmock:` section of **project.yml**. We can add an `:includes`: setting like this:

```
:cmock:
  :mock_prefix: mock_
  :when_no_prototypes: :warn
  :enforce_strict_ordering: TRUE
  :plugins:
    - :ignore
    - :callback
  :treat_as:
    uint8:    HEX8
    uint16:   HEX16
    uint32:   UINT32
    int8:     INT8
    bool:     UINT8
  :includes:        # This will add these includes to each mock.
    - <stdbool.h>
    - <stdint.h>
```

Enabling Mocks for Extern-ed Function Prototypes

With those include files added, let's run Ceedling again and get our next error:

```
projects\blinky> ceedling test:all

Test 'test_led.c'
-----------------
Creating mock for gpio...
WARNING: No function prototypes found!
Generating runner for test_led.c...
Compiling test_led_runner.c...
Compiling test_led.c...
Compiling mock_gpio.c...
Compiling unity.c...
Compiling led.c...
Compiling cmock.c...
Linking test_led.out...
build/test/out/led.o: In function `led_turn_on':
projects/blinky/src/led.c:10: undefined reference to `GPIOPinWrite'
```

Hmmm... we can't find `GPIOPinWrite` again. If we take a closer look, we can see a `WARNING: No function prototypes found!` message when trying to create the mock for **gpio.h**.

Again, this Tiva library is strange — this time because all the function prototypes in the header files are extern-ed:

```
extern void GPIOPinWrite(uint32_t ui32Port, uint8_t ui8Pins, uint8_t
ui8Val);
```

By default Ceedling/CMock won't mock functions labeled `extern`. We need to tell CMock to mock these functions by adding the `:treat_externs:` setting:

```
:cmock:
  :mock_prefix: mock_
  :when_no_prototypes: :warn
  :enforce_strict_ordering: TRUE
  :plugins:
    - :ignore
    - :callback
  :treat_as:
    uint8:    HEX8
    uint16:   HEX16
    uint32:   UINT32
    int8:     INT8
    bool:     UINT8
  :includes:
    - <stdbool.h>
    - <stdint.h>
  :treat_externs: :include # Now extern-ed functions will be mocked.
```

Hooray! Now our test will finally build, and we can actually run the tests:

```
projects\blinky> ceedling test:all

Test 'test_led.c'
-----------------
Creating mock for gpio...
Generating runner for test_led.c...
Compiling test_led_runner.c...
Compiling test_led.c...
Compiling mock_gpio.c...
Compiling unity.c...
Compiling led.c...
Compiling cmock.c...
```

```
Linking test_led.out...
Running test_led.out...

-----------
TEST OUTPUT
-----------
[test_led.c]
  - ""

--------------------
IGNORED TEST SUMMARY
--------------------
[test_led.c]
  Test: test_module_generator_needs_to_be_implemented
  At line (16): "Implement me!"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  0
FAILED:  0
IGNORED: 1
```

Add an Actual Unit Test

Setup and configuration is always a difficult part for embedded projects. Now though we've managed to fight through it... so that we can get down to the business of actually writing some unit tests.

Since the LED on our board is connected to pin 2 of port F, we might want to test that our `led_turn_on` function uses `GPIOPinWrite` to set pin 2 of port F. We can create a new unit test function in **test/test_led.c** and use an expectation to do this:

```
include "inc/hw_memmap.h"

void test_when_the_led_is_turned_on_then_port_f_pin_2_is_set(void)
{
    // Expect PORTF pin 2 to be set.
    GPIOPinWrite_Expect(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
```

```
    // Call the function under test.
    led_turn_on();
}
```

Note that we needed to `#include "inc/hw_memmap.h"` to get access to `GPIO_PORTF_BASE`, and `GPIO_PIN_2`.

And if we run our tests now, we can watch it pass:

```
projects\blinky>  ceedling test:all

Test 'test_led.c'
-----------------
Generating runner for test_led.c...
Compiling test_led_runner.c...
Compiling test_led.c...
Linking test_led.out...
Running test_led.out...

-----------
TEST OUTPUT
-----------
[test_led.c]
  - ""


--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
```

The Next Steps

Now that you have a unit test framework set up for your project, you're ready to start incrementally adding tests where you can — and gradually make your embedded software better.

You should consider trying to have tests for any new code you're writing, but do what you can. Are you chasing a bug? See if you can create a failing test for it first. Then make it pass. Bam, bug fixed!

Also — to make it easier to run the tests — you could set up your IDE to run `ceedling test:all` when you press a keyboard shortcut. Better yet, you could set it up to run `ceedling test:<your_current_file>` (with the current file in your editor). This only runs the tests for the file that your working on. Eventually when you have many more tests, this will be a lot faster.

References

Source Code

The source code used in these examples is available on GitHub.

Example 1	https://github.com/ElectronVector/try-tdd-with-ceedling
Example 2	https://github.com/ElectronVector/mocking-hardware-ceedling-cmock
Example 3	https://github.com/ElectronVector/add-ceedling-to-existing-project

Documentation

Ceedling	https://github.com/ThrowTheSwitch/Ceedling/blob/master/docs/CeedlingPacket.md
Unity	https://github.com/ThrowTheSwitch/Unity
CMock	https://github.com/ThrowTheSwitch/CMock/blob/master/docs/CMock_Summary.md

Ceedling Quick Reference

`ceedling new <project-name>`	Create a new Ceedling project named `<project-name>`.
`ceedling help`	Show all available Ceedling commands for the project.
`ceedling module:create[<module-name>]`	Create a new module named `<module-name>`. Creates a source file, header file and unit test file.
`ceedling module:destroy[<module-name>]`	Delete an existing module named `<module-name>`.
`ceedling`	Run all unit tests in the project (the same as `ceedling test:all`) this is the default operation.
`ceedling test:all`	Run all unit tests in the project.
`ceedling test:<module-name>`	Run only the unit tests for this module.
`ceedling clobber`	Delete everything created during the build of the tests. Like cleaning, but more.

Test Assertions

`TEST_ASSERT(condition)`	Pass if the condition is true.
`TEST_ASSERT_TRUE(condition)`	Pass if the condition is true.
`TEST_ASSERT_FALSE(condition)`	Pass if the condition is false.
`TEST_FAIL()`	Fail the test immediately.
`TEST_IGNORE()`	A test containing this statement is ignored.
`TEST_ASSERT_FLOAT_WITHIN(` `  delta, expected, actual)`	Pass if the two float value are within `delta` of each other.

`TEST_ASSERT_EQUAL_STRING( expected, actual)`	Pass if the two null-terminated strings match.
`TEST_ASSERT_EQUAL_STRING_LEN( expected, actual, len)`	Pass if the two strings match up to `len`.
`TEST_ASSERT_NULL(pointer)`	Pass if the pointer is a null pointer.
`TEST_ASSERT_NOT_NULL(pointer)`	Pass if the pointer is not a null pointer.
`TEST_ASSERT_EQUAL_MEMORY( expected, actual, len)`	Pass if the two regions of memory match.
`TEST_ASSERT_EQUAL_INT( expected, actual)`	Pass if the two signed integers match.
`TEST_ASSERT_EQUAL_UINT( expected, actual)`	Pass if the two unsigned integers match.
`TEST_ASSERT_EQUAL_INT_ARRAY( expected, actual, elements)`	Pass if the two arrays match.
`_MESSAGE`	`_MESSAGE` can appended to any of the other test assertions. Then there is on additional argument at the end of the argument list which is a string to be printed if the test fails.

For the complete list of test assertions, see the Unity documentation at: https://github.com/ThrowTheSwitch/Unity.

Mock Function Formats

Original Function	CMock Generated Expect Function
`void func(void)`	`void func_Expect(void)`
`void func(params)`	`void func_Expect(expected_params)`
`retval func(void)`	`void func_ExpectAndReturn(retval_to_return)`
`retval func(params)`	`void func_ExpectAndReturn(expected_params, retval_to_return)`

For the complete list of available mock functions, see the CMock documentation here: https://github.com/ThrowTheSwitch/CMock/blob/master/docs/CMock_Summary.md#generated-mock-module-summary.