The mock object approach to test-driven development

Jordan Schaenzle, Atomic Object - October 16, 2012

Editor's Note: *Jordan Schaenzle provides a guide to implementing test-driven development methods for your embedded systems using mock objects, whether or not you are doing your design in the C language or in C++.*

Test Driven Development (TDD) is growing in popularity as developers realize they produce better code and have greater confidence in their work when using TDD. Project managers are also discovering that TDD allows teams to maintain a more predictable, stable pace and greatly reduces the debugging phase that typically occurs at the end of the development cycle.

Unfortunately, in the embedded software world TDD practices aren't gaining much traction. This is a concern as the nature of embedded devices makes good coding practices all the more important. One "minor" bug could result in inconvenient firmware updates or a product recall, putting a significant scar on a company's reputation.

Some embedded developers believe it's not possible to use automated testing because their code has to interact with hardware peripherals. Also, a large percentage of developers are not actively writing new code, but instead are updating and improving existing code bases. They often believe that it's too late in the game to implement a testing process or that doing so would require them to change the entire structure of their project.

In reality, these worries are not based in fact and can be solved by an important piece of TDD called 'mocking', which enables developers to easily write tests that verify isolated chunks of code. Several excellent testing frameworks exist that are custom tailored to embedded development and facilitate the use of mocks.

As embedded applications grow in size they are often broken into subsections that we call modules (**Figure 1**). In many cases, a given module will make calls into one or more neighboring modules. Those modules, in turn, make calls into other modules and the chain continues all the way down to the lowest-level hardware interactions.

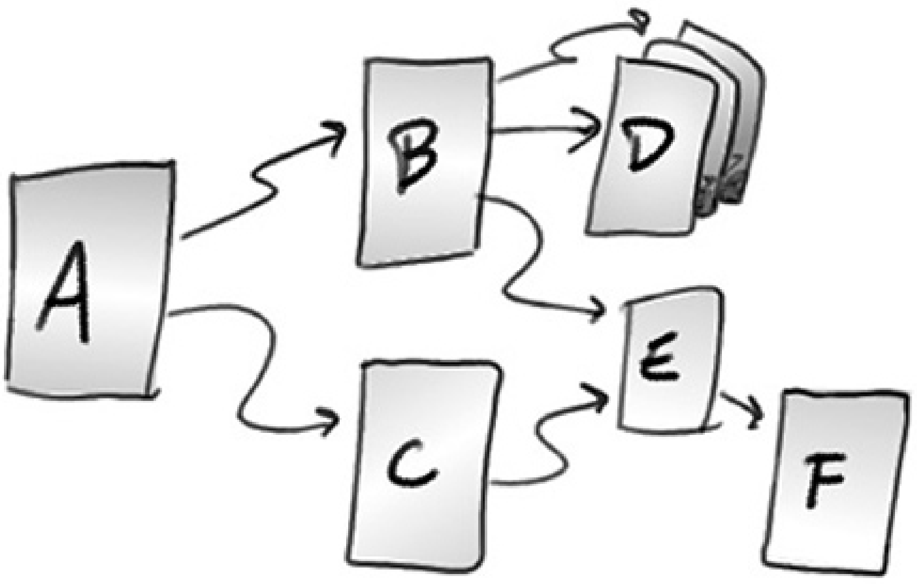

Figure 1. Typical application structure

This chain of interaction poses a problem when writing unit tests. How do you test to ensure a single module is doing what is expected, without also testing all of the modules that it touches? Attempting to specify a state for every module in the chain, for every test, is a huge undertaking. Tests would be lengthy, extremely brittle, and would need to be refactored every time something in the chain changed. 'Mocks' were invented to solve this problem.

What is a Mock?

A mock is a stand-in for a real module. It doesn’t contain any real functionality but rather imitates a module’s interface. When used in a test, a mock intercepts calls between the module under test and the mocked module (Figure 2). If the function being called has a return value, the mock for that function will also return a value as specified in the test.

Additionally, mocks have a special feature that keeps track of the number of times each function is called, and the order of those calls. If function calls occur in an unexpected order, the problem is detected and failure is triggered. Mocks are able to break the chain of interaction and isolate a given module so that it can be easily tested.

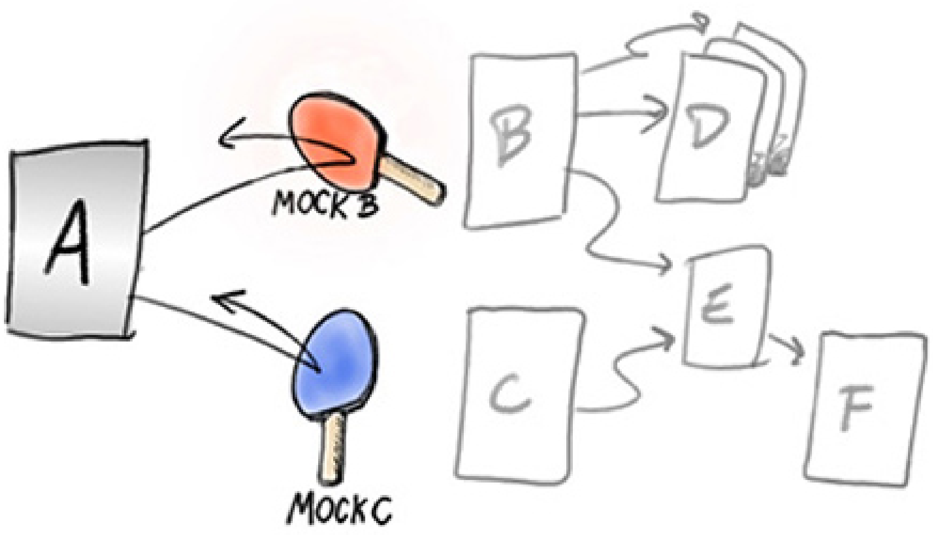

Figure 2. Mocks act like a ping-pong paddle by intercepting the function call and immediately returning execution to the calling function.

How are Mocks Created?

Right now, you might be thinking that creating these mocks would mean a lot of additional work. This is where mocking frameworks come into play. A mocking framework is a tool that generates mocks for you. The beauty of it is that you don't even need to write a module's implementation to generate a mock.

CMock, for example, is a free mocking framework (Figure 3) that is designed for use with C. To generate a mock, all it needs is a header file. CMock parses the function declarations in the header file and generates mock functions based on the prototypes. GoogleMock and CppUTest are also available for generating mocks for C++ projects.

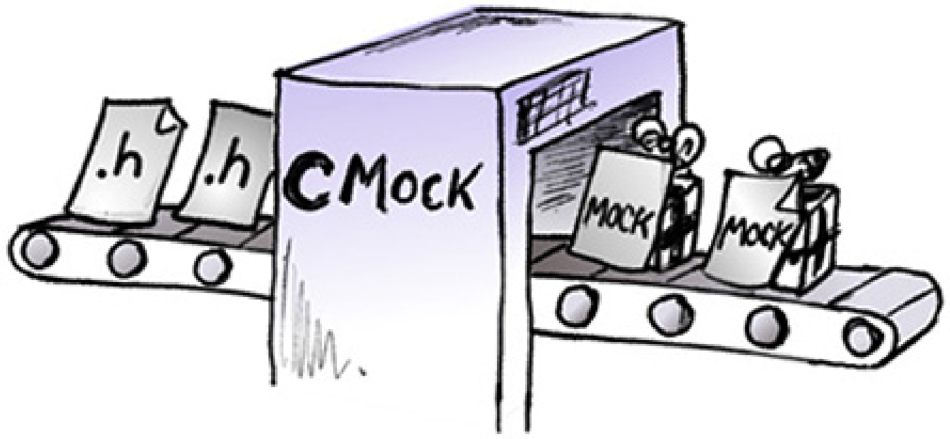

Figure 3. CMock creates mocks using only header files.

Testing frameworks and mocking frameworks are generally used together within a build automation environment. This integration allows for efficient use of these tools in an automated fashion. The tools work together to automatically detect when a mock is needed and generate it on the fly, during the build process. The mock objects and real objects can then be linked together to create test executables.

How are Mocks Used?

A mock by itself is not very useful. The true value is realized when you use a mock in a unit test. We'll walk through a simple demo, using TDD, to see how mocking comes into play. Explaining the entire TDD process is outside the scope of this article so I will focus primarily on how mocks fit into the overall process. I will be using a C unit testing framework called Unity for the tests, and CMock for the mocking framework. For a good primer on unit testing and an introduction to Unity and CMock, have a look at **throwtheswitch.org**. Also, to download the source code for the complete example see the References section below.

For this example, we'll use an LED controller module. To start out, we'll implement a single function that simply turns an LED on. The function will take a single parameter to indicate which LED should be turned on. When called, it will invoke a function in a GPIO module to set the correct GPIO pin state.

```
// test_LedControl.h
#include “LedControl.h”
#include “mock_Gpio.h”
void
test_LedControl_TurnLedOn_should_set_GPIO_pin_1_when_turning_on_
the_red_LED()
{
   // Setup expected call chain
   GPIO_SetPin_Expect(1U);

   // Call the function under test
   LedControl_TurnLedOn(LED_RED);
}
```

Listing 1 - Unit Test

Listing 1 is a simple unit test but it illustrates some important points. First, notice that we included `mock_Gpio.h` rather than the real `Gpio.h`. This tells the test framework to use a mock (fake) version of the Gpio module. Next, notice the first line of code in the test. This line tells CMock it should expect function `GPIO_SetPin` will be called with a single argument equal to 1 (the GPIO pin to set). The last line of code in the test tells the testing framework to call the function under test, `LedControl_TurnLedOn`, with a single argument equal to `LED_RED`.

It’s important to note that we didn’t check the state of the GPIO pin to verify that it actually changed state. This was intentional, as we are not testing the functionality of the Gpio module in this test. We are isolating the LedControl functionality and verifying that it interacts with other modules correctly.

The test in Listing 1 is complete but will most certainly fail for several reasons. First, the compiler doesn’t know what `GPIO_SetPin_Expect` is, because a mock cannot be created yet. In order for CMock to generate one we need to create a header file. Let’s do that now, as shown in Listing.

```
// Gpio.h
void GPIO_SetPin(uint8_t pin_num);
```

Listing 2 - GPIO Module Header File

That’s it! This function definition is all CMock needs to create the mock implementation of our GPIO module. The last thing we need to do is implement the LedControl module to satisfy the test.

```
// LedControl.h

#define RED_LED_PIN 1
#define BLUE_LED_PIN 2

typedef enum {
   LED_RED,
   LED_BLUE
} LED_T;

void LedControl_TurnLedOn(LED_T led);
```

Listing 3 - LedControl Header File

```
// LedControl.c
#include "LedControl.h"
#include "Gpio.h"
void LedControl_TurnLedOn(LED_T led)
{
   if(LED_RED == led)
   {
      GPIO_SetPin(RED_LED_PIN);
   }
}
```

Listing 4 - LedControl Source File

In **Listing 3** and **Listing 4** we have the header file and source file for the LedControl module. The header file contains an enumeration to represent the LEDs on the board, a couple of define statements to correlate a GPIO pin to each LED, and a prototype for the function under test. The source file contains the actual definition of the function under test.

When the testing framework builds the test in **Listing 1**, it will use the real LedController module. However, because we included `mock_Gpio.h` in the test file, it will link in the mock version of the Gpio module generated by CMock. When the test runs, the real `LedControl_TurnLedOn` function is called. This function calls `GPIO_SetPin`, which exists in the mock object. Because we set up the expectation in the first line of our our test, the mock object will validate that the argument used in the call matches that which was specified, and return happily. If the `LedControl_TurnLedOn` function were to call into the GPIO module again, or if the argument's value was something other than 1, the mock object would produce an error and the testing framework would report it in the test summary.

This example demonstrates the TDD process for developing entirely new code. If you are working on legacy code, TDD can still be used effectively. Write a test to the existing code and make sure it fails. Then, change the code as needed until the test passes. Use mocks to isolate the module you are updating in the same way we did above. Occasionally, this will require some refactoring but as a side benefit you will end up with a better, object-oriented program structure.

Page 2 of 2

Why Mocks Improve Development

Mocks play a critical role in the TDD process, making it possible to develop code using a top-down approach. This means that you start at the beginning of the program, the main loop, and work your way down to the lowest-level peripheral drivers. When you get to a point where you need a function, or a module that doesn't exist, you simply mock it out and continue with the current module. Developing this way has several distinct advantages. First, it allows each module's API to evolve naturally. No time is wasted coding features which may go unused. Second, it saves you from having to predict the interface a given module will need ahead of time. The growth of each interface is constrained to exactly what is needed by the rest of the application.

Mocks also improve the development process by facilitating a methodology known as interaction-based testing. When using interaction-based testing the developer need not be concerned with the internal state of objects. The tests simply verify that a given module interacts with other modules in an expected way and produces an expected result based on a given set of inputs. In the example above, remember that when we tested the LED controller we did not assert that any output pins were actually turned on. We did not concern ourselves with the resulting state but rather verified the interaction of the LED module with the GPIO module.

TDD and Hardware Interaction

Many embedded developers resist using TDD because of the hardware interactions that are inherent to embedded applications. How do we effectively get around this problem? The answer is by using encapsulation (Figure 4). Let's say, for example, your device communicates with a PC via a UART peripheral. A smart way to approach this would be to create a UART driver. The UART driver would consist of basic functions like SetBaudRate(), SendString(), and ReadLine(). This way, all of the device specific code is contained within the UART driver. If you ever have to change microcontrollers, all the code you need to change would be isolated in one module. Then, the rest of your application can be easily tested based on its interactions with the driver. Developing this way leads to a well structured, object-oriented application.

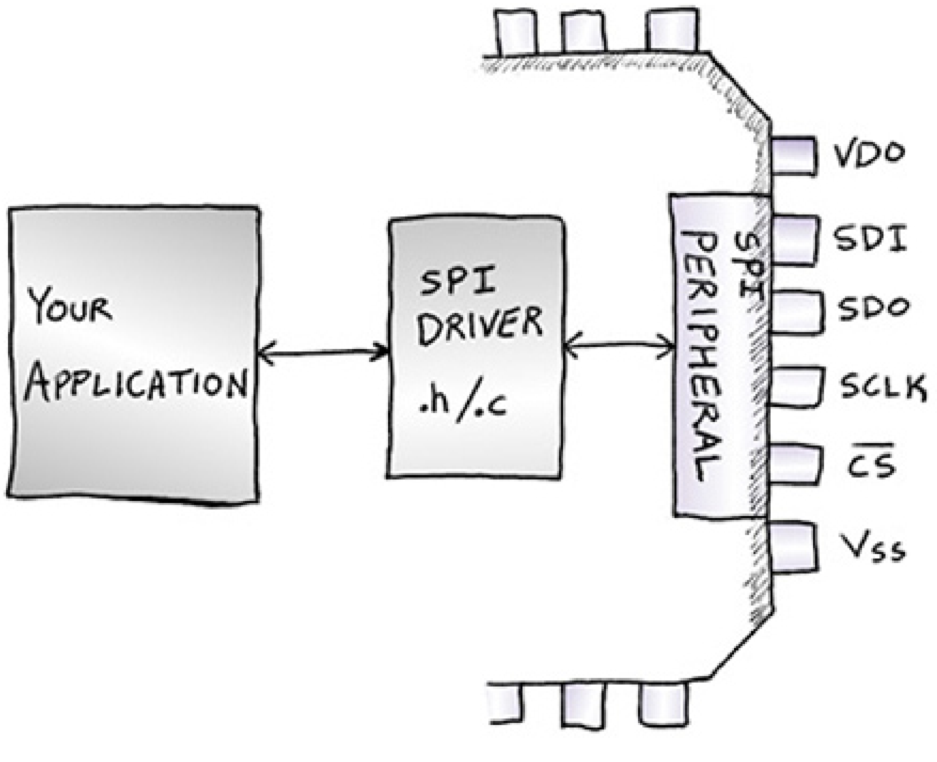

Figure 4. Hardware interactions can be tested by isolating device specific code in a driver which can be easily mocked.

What About Special Function Registers?

As you may have guessed, there comes a point when there is nothing left to mock. For example, if we were to write tests for our GPIO module above, how would we test that the correct output pin is actually being turned on? This is where other components of the testing framework come into play.

In general, hardware peripherals are controlled via Special Function Registers (SFRs). Ideally, we should have tests to verify that these registers are set correctly. The way in which we verify this will depend on how our testing framework is configured. For example, if the framework is set to run tests on a simulator, the SFR values can be checked in the same way as normal variables.

However, if a simulator is not available, a common alternative is to create an SFR header file. This header file will contain dummy variables that match the SFRs available on the device. You test against these dummy variables as if they were the real SFRs. For a more concise explanation, read James Grenning's book, **Test Driven Development for Embedded C**.

Is TDD Really Worth My Time?

The demand for products with embedded electronics is exploding. This increased demand requires embedded developers to work faster and produce more without sacrificing the quality of their final product. TDD makes this possible by increasing the speed and accuracy of development. Will TDD take time to learn? Yes. Will it change the way you write code? Yes. Is it worth it? Without a doubt.

Not only do you get instant verification of your code using TDD, but you also have a well organized code base as a result. In addition, you eliminate nearly all of the time-consuming, on-target, IDE-based debugging that would otherwise be necessary. Utilizing Test Driven Development helps developers produce higher quality work and gain more confidence. Higher confidence equates to less stress, greater productivity, and a better final product.

Jordan Schaenzle *is an embedded software specialist at* ***Atomic Object*** *in Grand Rapids, MI. He earned his BSE with a concentration in Electrical Engineering at Calvin College in 2007. Prior to working at Atomic Object, Jordan designed electronics for automated astronomy equipment.*

Resources:
1 - **Complete Example Source Code**
2 - **Throw The Switch**
3 - **Test Driven Development for Embedded C**, by James Grenning
4 - **CppUTest**
5 - **Google Code**
6 - **Google Mock**