Testing Special Cases

Introduction: The following began life as a collection of blog posts written by Mark on his personal blog. These sections cover a variety of special situations that you will almost certainly encounter when trying to unit test real embedded and systems-level C code. While not a complete list of all challenges you may encounter, the following sections provide at least one perspective on how to handle a variety of tricky “corner cases” in unit testing.

Mocking: Some of the sections that follow reference mocks and the CMock tool. At this point in the course we have not yet addressed mocking in Test-Driven Development of embedded C (but this will be the next major expansion of the course). So let’s briefly summarize mocking for now just to give you a flavor for it.

Your source code calls other source code. In unit testing you want to isolate a given source module to exercise it thoroughly. So how do we handle the source modules our code under test interacts with outside itself? What we do is fake those other modules. A mock has the same interface as a source module you would like to simulate. In C, through the magic of linking, we can swap the compiled mock object in for the real source object—so long as the mock has the same function signatures our linker won’t care.

CMock is a tool that takes as input the header file of a source module and generates all the C code needed for mock simulation behind the interface. With the features that are generated within this mock we can tell the mock module how often we expect certain functions to be called by our source under test, what they should return, and what to expect as input parameters. This allows us to simulate the real modules surrounding a source module under test. In our test code we declare which mocks to use, set up the expectations for these mocks, and, after the source under test is exercised, we ask the mock to compare the expected behavior with the actual. For more explanation and code samples, see here, here, and here.

Again, we’ll cover mocking in detail in the material we next add to the course.

Special Cases Addressed in this Document:

- main()
- Infinite Loops
- Private Data
- Changing Register Values
- Multiple Writes to the Same Register

main()

Let’s talk about testing that ubiquitous function in C: `main`. If our unit test files declare a `main` function of their own, how can we make them call our release code's `main` function?

It's a simple little hack, actually. We're going to use our old friend, the `TEST` define. If we always build our tests with `TEST` defined, then we can count on this to give us a little help.

In main.c, we add a few lines at the top, like so:

```
#ifndef TEST
#define MAIN main
#else
#define MAIN testable_main
#endif
```

Now, instead of declaring our main function as `main`, we use our macro… like this:

```
int MAIN(int argc, char* argv[])
{
    //Stuff worth testing...
}
```

When we write our test, we can now call `testable_main` in place of `main`, and proceed with unit testing as usual. `testable_main` won’t conflict with the `main` in our test files but will revert to our old friend `main` in our release build.

Our `main` in our source file is likely going to be coordinating many other modules... so in all likelihood, we'll write our test with mocks set up to expect the calls `main` makes outside itself.

Infinite Loops

While infinite loops might strike fear into the hearts of many a software developer, we embedded software types know that they are just darn useful in the right place. We use them to control our main loops. We use them to trap errors upon which we want to halt everything. We use them to build realtime operating systems. Sometimes a well-placed infinite loop is the best tool for the job.

But, how do we test such a thing? If the loop is indeed infinite, it's never going to return for our test to actually finish. Clearly, a little hack is required.

If your mind jumped to our friend the `TEST` macro, your intuition served you well. If not, well, we'll keep working on it. We like to define the keyword `TEST` when compiling all our tests. This allows us to make an occasional tiny change in our release code. We try not to abuse this power... after all, if our code changes much during a test, it's not really a valid test, right?

But, an infinite loop is a great place to apply our powerful `TEST` define friend. We're going to create a macro called `FOREVER`. During a release, `FOREVER` is going to be defined as 1. This allows us to easily make infinite loops, like this:

```
void InfiniteLoopBelow(void)
{
    while(FOREVER())
    {
        //do stuff until the batteries die
    }
}
```

During a release, this translates to `while(1)` which will always evaluate to true, and will therefore always run forever.

But what do we do with this macro during a test? Here are three different ways you could use it to solve your problem. Choose the one that fits best for you.

RUN ONCE

The simplest approach is to run our loop just once during tests. This is fairly straightforward. Instead of structuring the loop as we did above, we use this form:

```
do {
    //do stuff until the power company shuts us down
} while (FOREVER());
```

When our loops are shaped like this, we can write tests that just redefine `FOREVER` to be 0 during a test. This will cause us to only run the loop once.

```
#ifndef TEST
#define FOREVER()  1
#else
#define FOREVER()  0
#endif
```

A COUNTER

A more flexible solution is to replace our `FOREVER` call with a post-decrementing variable. Something like this:

```
#ifndef TEST
#define FOREVER()  1
#else
extern int NumLoops;
#define FOREVER() NumLoops--
#endif
```

Then, in our test, we declare an instance of this variable. We can then sprinkle it about in our tests as needed.

```
int NumLoops;

void setUp(void)
{
    //Assume no loops unless we specify in a test
    NumLoops = 0;
}
```

```
void tearDown(void)
{
    TEST_ASSERT_EQUAL(0, NumLoops, "Forever called unexpected number
of times);
}

void test_SillyStuff(void)
{
    NumLoops = 5;
    CallFuncUnderTest();
    //It should iterate over the internal FOREVER loop 5 times
    //Otherwise tearDown will throw an error for us
}
```

This is a useful and simple way to control the number of times we execute a loop. It also gives us the ability to verify that the loop was called as many times as we expected (i.e. ensuring we secretly exit the loop early somehow).

CMOCK

The most advanced technique is to use CMock. Like the other methods, it still involves using our handy macro. This time, though, we put our macro in a mockable release header file.

```
#ifndef TEST
#define FOREVER()  1
#else
int FOREVER(void);
#endif
```

As you can see, if we have `TEST` defined, we have a function prototype. If we tell CMock to mock this file, CMock is going to notice this function prototype and create a mock for it. It doesn't matter that under normal circumstances `FOREVER()` always returns 1... in a C code operation, a lone integer is equivalent to the returned integer of a function. Now that the function is mocked, we have full control of what it returns through CMock.

```
#include "MockUtils.h"

void test_SillyStuff(void);
{
    FOREVER_ExpectAndReturn(1);
    //Other Expectations and Stuff
    FOREVER_ExpectAndReturn(1);
    //Other Expectations and Stuff
    FOREVER_ExpectAndReturn(0); // loop exits
    //Expectations for Anything AFTER the Loop!

    CallFunctionUnderTest();
}
```

Private Data

In C, “private” data often takes the form of module-scoped variables. These variables are meant to be shared amongst the functions in our module but not seen by the outside world. Private data falls into two categories:

SOFT PRIVACY

If our privacy is soft, it means our variables are declared in the C file, but aren't really protected... so they are effectively global if anyone cares to `extern` them. Of course, everyone knows they *shouldn't* `extern` them into their file and use them directly, but since we didn't actually protect them, we can't get too angry when this promise is broken.

HARD PRIVACY

We implement hard module privacy through the innocent-looking `static` keyword. Once we declare that variable as `static`, its name doesn't leave the current module and can't be accessed without jumping through serious hoops. Selectively using `static` is a good practice to ensure that those accessing our modules are accessing only the data and methods that we want them to have access to.

CAN WE TEST IT?

Well, clearly we *can* test things that are soft private. All we need to do is use `extern` in our test. Boom. We have access to anything we want. As we'll discuss in a moment, we *can* also test hard private data and functions.

Before we discuss that, though, we should talk about a bigger question: *should* we test it? Needing access to internal data is often an indicator that our module is not built as well as it could be. It's a code smell. It doesn't necessarily mean we've screwed up... but it's an indicator we might have.

Before using a skeleton key to give ourselves access to private data, we should see if there is a cleaner way to test the module. Can we treat it more like a black box? Can we refactor the module itself to be cleaner?

Still convinced we need to get at the data? Occasionally it does so happen that this is the best way. So let's do it.

FABRIC SOFTENER

Our skeleton key in this case is fabric softener! We want to turn hard private data into soft private data. If we control the source and the test, this is actually very simple.

```
#ifndef TEST
#define STATIC static
#else
#define STATIC
#endif
```

Instead of declaring our module-scoped variables with `static`, we instead declare them as `STATIC`. Our scoping rules are still applied when compiled for release, but the hard privacy is dropped when compiled for a test. We can then reach in using `extern` and read/write the data as needed.

ACCESS GRANTED

That's all there is to it. One particularly useful application of this is to use private data to build up particularly complex modules in a test driven fashion (a state machine, for example). Once the module is built, we write tests that do not use private data and drop the temporary tests used to build the module (thankfully our test suite will tell us if we've broken anything as we update our tests).

Changing Registers

How do we wait on a flag to be set during a test? How do we read from a single-address queue register? It's often not obvious how to test complex register features.

Let's start with something we already know how to test, then build up a couple of more challenging examples. Here is a simple function:

```
int SimpleFunc(void)
{
    if (PORTA_DIR & 0x1000)
        if (PORTA_OUT & 0x1000)
            return 1;
    return 0;
}
```

It might have tests that look something like this:

```
void test_SimpleFunc_should_return_0_IfWrongDirection(void)
{
    PORTA_DIR = 0x2000;
    PORTA_OUT = 0x1000;

    TEST_ASSERT_EQUAL(0, SimpleFunc());
}

void test_SimpleFunc_should_return_0_IfWrongValue(void)
{
    PORTA_DIR = 0x1000;
    PORTA_OUT = 0x0001;

    TEST_ASSERT_EQUAL(0, SimpleFunc());
}

void test_SimpleFunc_should_return_1_IfBothRight(void)
{
    PORTA_DIR = 0x1000;
    PORTA_OUT = 0x1000;

    TEST_ASSERT_EQUAL(1, SimpleFunc());
}
```

Tests like this are easy to set up because they only involve one read of each register. Most often this will get us by... but occasionally, we need to check the same register more than once. We could break it into multiple functions, but sometimes they just logically belong together. So what do we do then?

Let's start with an example.

```
int FetchMessage(char* data, int len)
{
    int i;

    if (len > 8)
        return ERR_TOO_LONG;
    if (len < 1)
        return ERR_TOO_SHORT;

    if (!(COMM_STATUS & COMM_OK_TO_RECV_FLAG))
        return ERR_CANT_RECV;

    for (i=0; i < len; i++)
    {
        if (COMM_STATUS & COMM_BUFFER_EMPTY)
            return i;
        data[i] = COMM_DATA_IN & 0x00FF;
    }
    return len;
}
```

There are a couple of things to notice about this function. First, we read from `COMM_STATUS` at the beginning to ensure we are configured to receive data and then for each incoming byte to verify our peripheral's queue isn't empty.

The second thing to notice is that we receive data through reading the `COMM_DATA_IN` register... which means we read it multiple times.

Let’s start by writing the tests that are reachable without jumping through any hoops at all. Often, you'll find that you can test a lot more than you think you can, just by being careful about how you preload your registers. These tests are a bit more brittle than usual, but they'll get you by in many cases:

```
char actual[8];

void setUp(void) {
    int i;
    for (i=0; i < 8; i++)
        actual[i] = 0;
}

void test_FetchMessage_should_ReturnErrorIfNotOkToRecieve(void)
{
    COMM_STATUS = 0;

    TEST_ASSERT_EQUAL(ERR_CANT_RECV, FetchMessage(actual, 8);
}

void test_FetchMessage_should_ReturnErrorIfBufferEmpty(void)
{
    COMM_STATUS = COMM_OK_TO_RECV_FLAG | COMM_BUFFER_EMPTY;

    TEST_ASSERT_EQUAL(ERR_CANT_RECV, FetchMessage(actual, 8);
    TEST_ASSERT_EQUAL(0, actual[0]);
}

void test_FetchMessage_should_GetASeriesOfData(void)
{
    char* expected = 'AAAAAAAA';

    COMM_STATUS = COMM_OK_TO_RECV_FLAG; //and NOT COMM_BUFFER_EMPTY
    COMM_DATA_IN = 'A';

    TEST_ASSERT_EQUAL(8, FetchMessage(actual, 8) );
    TEST_ASSERT_EQUAL_STRING(expected, actual);
}
```

So, we can get quite a bit of work done by not changing our registers mid-test, but we're stuck making assumptions. For example, we had to assume there is always data ready to get any data, and that the data is always the same value (`'A'`). Often, these assumptions are going to be an acceptable tradeoff. When they're not, it's often an indicator that we could do a cleaner job of deciding what work gets done where.

Occasionally, though, we might want to more thoroughly test such a feature and we're not in control of the implementation. In these cases, we can create more complex “register” definitions. Consider this:

```
#define RETURN_NEXT(vals, index)                    \
    ((vals == NULL) ? 0 :                           \
     ((--index > 0) ? vals[index] : vals[0])        \
    )

EXTERN int* COMM_STATUS_vals  = NULL;
EXTERN int  COMM_STATUS_index = 0;
#define COMM_STATUS RETURN_NEXT(                    \
                     COMM_STATUS_vals,              \
                     COMM_STATUS_index              \
                    )

EXTERN int* COMM_DATA_IN_vals  = NULL;
EXTERN int  COMM_DATA_IN_index = 0;
#define COMM_DATA_IN RETURN_NEXT(                   \
                      COMM_DATA_IN_vals,            \
                      COMM_DATA_IN_index            \
                     )
```

These “registers” are now set up to handle a queue of data that you set up ahead of time.

For example, if we wanted to create a test that verified we could receive three bytes of valid data, and then return, it might look like this:

```
void test_FetchMessage_should_GetDataUntilEmpty(void)
{
    char* expected = 'AAAAAAAA';
    int statuses[] = {
        COMM_OK_TO_RECV_FLAG,
        0,
        0,
        0,
        COMM_BUFFER_EMPTY,
    };
    int reads[] = { 'H', 'i', '!' };

    COMM_STATUS_val     = statuses;
    COMM_STATUS_index   = sizeof(statuses);
    COMM_DATA_IN_val    = reads;
    COMM_DATA_IN_index = sizeof(reads);

    TEST_ASSERT_EQUAL(3, FetchMessage(actual, 8) );
    TEST_ASSERT_EQUAL_STRING(expected, actual);
}
```

So this is much more flexible... but with this much complexity, we clearly don't want to test things this way regularly. We'll save this one for special occasions.

Multiple Writes to the Same Register

Now let's consider how to verify multiple writes to the same register from within the same function. For example, we might have a register into which we can queue characters as a means of sending data over a USART.

We're going to build on the foundation of the previous section. Let's put together an example of a function that accepts a character pointer, and writes those characters one at a time to a USART queueing register, then tells the peripheral to transmit all queued data.

```
int SerialWrite(const char* data, int len)
{
    int i;

    if (len < 1)
        return ERR_TOO_SMALL;
    if (len > 16)
        return ERR_TOO_BIG;

    for (i=0; i < len; i++)
        USART_OUT = (uint32_t)(uint8_t)(data[i]);

    USART_CMD = USART_SEND_FLAG;
    return len;
}
```

Much like in the previous section, we can create an enhanced “register” to store values written to it and then verify them after our source function call.

```
#define TEST_MAX_QUEUE 16
#define ENQUEUE(vals, index)                \
    vals[ (index > TEST_MAX_QUEUE) ?        \
          TEST_MAX_QUEUE :                  \
          index                             \
        ]

EXTERN char USART_OUT_queue[TEST_MAX_QUEUE+1];
EXTERN int  USART_OUT_size = 0;
#define USART_OUT ENQUEUE(                  \
          USART_OUT_queue,                  \
          USART_OUT_index                   \
        )
```

With this setup, we should be able to reset our counter before each test, then accept up to `TEST_MAX_QUEUE` values written to our “register”.

```
void setUp(void)
{
    int i;
    for (i = 0; i < TEST_MAX_QUEUE; i++)
        USART_OUT_queue[i] = 0;
    USART_OUT_index = 0;
}

void test_SerialWrite_should_EnqueueAShortString(void)
{
    char* expected = 'Awesome';
    int len = strlen(expected);

    TEST_ASSERT_EQUAL(len, SerialWrite(expected, len) );
    TEST_ASSERT_EQUAL(len, USART_OUT_index);
    TEST_ASSERT_EQUAL_STRING(expected, USART_OUT_queue);
}
```

Like the situation where we were queueing up results to read from registers, this is a lot of complexity just to test something. While there will be situations where this sort of thing is useful, more often we will find that it suggests we have made an interface that is more complex than it should be.

If we get to this level of complexity, we should likely consider using CMock for these cases instead. But, when the situation demands it, we can now handle it.