How Do I Test That?

This is a collection of commonly asked testing questions of the years. Many of these come from Mark's personal blog on testing, but have been collected here for your convenience. We hope this helps in your unit testing endeavors!

main()

Let's start by talking about testing that ubiquitous function in C: main. If our tests 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're going to 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're going to use our macro. Something 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 test as usual. Being the main function, it's likely going to be coordinating many other modules... so in all likelihood, we'll be wanting to dust off our CMock skills and get down to work.

Infinite Loops

Next let's talk about infinite loops. While they 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 that we want to halt

everything for. We use them to build realtime operating systems. Sometimes a well-placed infinite loop is the best tool for a 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 for all of 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.

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 make it so we always run our loop contents only 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
```

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 wrong # 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 expected (that we didn't secretly get another a way out early).

CMOCK

The most advanced technique is to use CMock. Like the others, 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 mock for it. It doesn't matter that under normal circumstances FOREVER() always returns 1... an integer is just like a function that returns that int, just more efficient. 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);

  //Expectations for Anything AFTER the Loop!

  CallFunctionUnderTest();
}
```

There we are. Three different approaches to solving the infinite loop problem. There likely are others, but these three have served us well.

Private Data

Let’s talk about private data. Private data often takes the form of module-scoped variables. They're meant to be shared amongst the functions in our module, but not to be seen by the outside world. The 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, it's name doesn't leave the current module and can't be accessed without jumping through serious hoops. This is a good practice to make sure that those accessing our modules are accessing just the data and methods that we want them to have access to. Everything else should be defined as static.

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? It DOES 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 built, we write tests that don't use the private data to test the overall system behavior. We can now drop the temporary tests.

Registers

Let’s tackle the jumbo topic of registers. This is THE most common question people have when they start thinking about unit testing embedded software. It's the reason it's better to test in a simulator or using native code than it is to test on the target hardware (Not familiar with this discussion? Catch up here).

We begin this tale with good news and bad news.

GOOD NEWS, THE FIRST

First, if we're using a simulator, we have really good news: We're probably done! Simulators almost always treat registers as just a big pool of RAM. That's perfect! We can write initial conditions or testable values in our tests. Our source will then read and write to them normally. Finally, our tests can again read from them to verify our source did what was expected. It

doesn't matter if the target considers those registers as read-only, write-only, or whatever... we can simulate whatever we want.

GOOD NEWS, THE SECOND

Second, even if you are using a native application, transforming your register set into something testable isn't hard. But...

BAD NEWS

It's tedious. Most modern microcontrollers have complex register sets... and we need to do some footwork to prepare for testing any of the ones that your application may require!

Our goal? Our goal is to "remap" all those registers regular heap-declared variables so that we can read/write them as desired. The address 0x10000000 might be the GPIO control register on our target hardware, but it's unlikely to be directly accessible under Linux / Windows / Mac OS / whatever.

The trick is to rebuild the micro's header file, transforming it into something testable. It's straightforward... we want to use the define TEST to know if we're creating a RAM version of those names, or if they will continue to be mapped to the register set. The details really only vary in how our supplier built their micro's header file in the first place. Let's look at some examples... enough to give us an idea of how to approach other variants.

THE INT POINTER REGISTER

One common paradigm (if not THE most common paradigm) is to treat registers as some form of integer (usually unsigned) and access them through dereferencing a pointer to a specific address. It looks something like this:

```
#define PORTA (*(volatile uint32_t*)(0x40001000))
```

There are many slight variations on this theme, which might change the size of the int, use macros to make it easier to understand, or other such tricks. The point is that we're casting a memory location to a pointer, which we then dereference and treat as some sort of integer.

These are easy to create testable alternatives, because it's already telling us what type it wants to be. We just need to pay attention. We then create an alternate standin that we can use during a test:

```
#ifndef TEST
#define PORTA (*(volatile uint32_t*)(0x40001000))
#else
EXTERN volatile uint32_t PORTA;
#endif
```

We should keep in mind that the address isn't important when running the test. As long as all accesses happen through a consistent interface like this, the tests and the releases will both be happy with our definition of PORTA and everything should work well.

Occasionally, we come across registers that overlap. In these situations, we can use defines and casts to mimic the same behavior.

```
#ifndef TEST
#define SPI_WORD (*(volatile uint16_t*)(0x80000000))
#define SPI_HI (*(volatile uint8_t*)(0x80000001))
#define SPI_LO (*(volatile uint8_t*)(0x80000000))
#else
EXTERN volatile uint16_t SPI_WORD;
#define SPI_HI (*(volatile uint8_t*)(void*)(&SPI_WORD+1))
#define SPI_LO (*(volatile uint8_t*)(void*)(&SPI_WORD))
#endif
```

The (void*) is useful to keep your compiler from complaining about the pointer casting.

THE STRUCT POINTER REGISTER

Another common paradigm found in header files is the use of custom structs. Often a struct is created for a single peripheral on the microcontroller. Then, one or more memory addresses are cast to be "instances" of that struct. It's effectively just a fancier version of the INT POINTER case, and the solution is the same:

```
typedef struct _GPIO_PORT_T {
  volatile unsigned int DIR;
  volatile unsigned int PULLUP;
  volatile unsigned int INPUT;
  volatile unsigned int SET_OUTPUT;
  volatile unsigned int SET_INPUT;
} GPIO_PORT_T;
```

```
#ifndef TEST
#define PORTA (*(GPIO_PORT_T*)(0x40001000))
#define PORTB (*(GPIO_PORT_T*)(0x40002000))
#define PORTC (*(GPIO_PORT_T*)(0x40003000))
#else
EXTERN GPIO_PORT_T PORTA;
EXTERN GPIO_PORT_T PORTB;
EXTERN GPIO_PORT_T PORTC;
#endif
```

As you can see, it quickly becomes very similar to our first example.

THE COMPILER EXTENSION

Sometimes the developers of compilers for embedded applications give you "helpful" extensions to the C language. It's tempting to use these conventions because they provide shortcuts to otherwise tedious tasks... but often they are just keeping your code from being as portable as it could be.

One extension that seems to show up from time to time is the use of the *commercial at* (this guy: '@') to lock a variable to a location. It effectively does exactly what our two options above do, but with a neater notation, something like this:

```
volatile uint32_t PORTA @ 0x400000000;
```

That's kinda nice to look at, right? It is, but it's not standard C. If you try to compile your tests using a native compiler, it is likely to complain about these extensions.

Depending on your compiler, you may be able to get around this just by properly using your `#ifdefs`, and the compiler will ignore all your release-only code. You can then use one of the tricks above.

Otherwise, we need to get a little fancier: We need to create a second header file.

THE IMPOSTER HEADER FILE

We've shown all the examples above using our old friend `#ifdef` and the `TEST` define. That's often a good way to go, but maybe we don't want to use that trick? Micro definition files are often already huge, why make them bigger? Maybe we're not liking the way that this has required a lot of changes to our "release code" (even if it's all in opposite `#ifdefs`).

Surely there is another way!

There is. We make a second header file. We name it EXACTLY the same thing, but put it in a different folder in our source tree (possibly directly in our test folder, so that it's clear it goes with our tests). We fill it with the testable versions of all our registers. Then, we tell our build system to use that header for tests, and the other header for releases. Done. A little organization goes a long way!

EXTERN

Did you notice that all the testable examples above have an all-capital EXTERN in front of them? Did you figure out why?

The thing about micro register files is that they tend to get included a lot. Our test likely will include it, as will the release file we are testing. It's possible that other files getting compiled into the mix will also be including this file.

This is harmless when it's full of macros that are just dereferencing pointers... but now that we are filling it with actual instances of variables, we're going to start getting multiple declarations of things. That's going to be a problem when we link!

Because of this, we employ this little trick. At the top of our register file, we include this:

```
#ifdef TEST
#ifndef EXTERN
#define EXTERN extern
#endif
#endif
```

So, if we're building a test and we haven't defined EXTERN yet, we set it to extern. So far, that's all files, right? So now we just need to have ONE of our files in each test define EXTERN to be nothing, so that it will declare an actual instance of these variables.

The best candidate is the Test file, since there is only one Test file per build. You can do it the obvious way.

```
#include "unity.h"
#define EXTERN
#include "testable_micro_registers.h"
#include "file_to_test.h"
```

Or, a little more tidy, you can create a very small header file which looks like this:

```
#ifndef TESTABLE_MICRO_REGISTERS_H
#define TESTABLE_MICRO_REGISTERS_H
#define EXTERN
#include "micro_registers.h"
#endif
```

Then, you can include that INSTEAD of the normal micro register header in your tests only, like so:

```
#include "unity.h"
#include "testable_micro_registers.h"
#include "file_to_test.h"
```

Faking 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? Many architectures have registers that have some complex behavior, and it's often not obvious how to test it at first.

We're going to build on the foundation of what we already know about testing registers. If you would like a refresher, you can find it here.

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 make sure 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.

First, we 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 intCOMM_STATUS_index = 0;
#define COMM_STATUS RETURN_NEXT( \
    COMM_STATUS_vals,            \
    COMM_STATUS_index            \
  )

EXTERN int* COMM_DATA_IN_vals= NULL;
EXTERN intCOMM_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.

Verifying Multiple-Write Registers

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

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 send it all.

```
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 our recent discussion about queueing up data to send, we should try a queue. A good solution is to store written values into a register and then verify them after the function call. We'll again use registers for this.

```
#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 intUSART_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, it is likely we should be considering using CMock for these features instead. But, when the situation is right, we now can handle it.

Final Word

The great thing about being part of a community of like-minded individuals is that you have a resource for any issues you may encounter. If you run into a challenging test case, it's very likely others have also run into a similar issue. So please feel free to make use of the forums at http://throwtheswitch.org/forums. They're free!