Continue

C unit testing

This article is the series on Unit testing in C and carries the discussion on Unit Testing and its implementation. The aim of this series is to provide easy and practical examples that anyone can understand. This is the Introduction of Unit testing – Unit testing in C tutorial Part 1. You can also read ceedling installation, Unity, Cmock, and stringizing, token pasting in C. Unit Testing in C – Introduction Anyone who has been involved in the software development life cycle (SDLC) for a while, will have encountered some form of testing. Software testing is an acceptance mechanism for discovering how well a software works according to the specified requirements. Although the aim of testing is to find bugs, it cannot guarantee the absence of other faults, no matter how creative the test cases have been designed. Unit testing enables a more thorough level of acceptance testing. What is Unit Testing? A unit is simply a small piece of code for any single function. So when we test those units, it is called a unit test. The unit test is a short script or piece of code designed to verify the behavior of a particular unit independently to produce a pass or fail result. Unit Testing is performed during the application development phase. Unit testing is performed usually by developers. In V-model, SDLC, and STLC the unit testing is the first phase of testing before integration testing. It is a white box testing technique and QA engineers can also perform Unit Testing if required. However, it can sometimes be quite difficult to write a good unit test for a particular piece of code. Having difficulty testing their own or someone else’s code, developers often think that their struggles are caused by a lack of some fundamental testing knowledge or secret unit testing techniques. In this unit testing tutorial series, I intend to demonstrate that unit tests are quite easy; the real problems that complicate unit testing. Why is Unit Testing necessary and its use? Sometimes, developers skip out unit testing due to lack of time. Skipping out unit testing leads to more defect fixing costs during integration, Beta, and system testing. Proper unit testing at the time of application development, saves time and money. Here are the key reasons to perform Unit Testing. The defects can be fixed early at the development stage, and it saves time and costs. It helps developers to understand the code base and make changes quickly that you think of. Good unit tests generally serve as the project documentation. Unit tests can be reused and migrated to the new project quickly when required. You should tweak the code a bit to run again. Better Design – When developers write unit tests, their emphasis is on thinking about how their code will be used throughout the system, which generally results in better design. Unit Test on Embedded Software/Firmware Unit tests can help you write a better embedded software. To allow unit testing for a software project, the R&D team must write a testable, modular code – code that can be divided into self-contained units that can be tested. On top of making the code testable, embedded software developers must make sure their code is portable. The unit test won’t test the functionality like how it is running in hardware. Do we need hardware (target) to run the unit tests? Not really, if you use ceedling framework for unit tests, then you don’t need your target hardware. You can use your PC to run the unit test. In our full unit test series, we are going to discuss ceedling only. So your PC machine is enough to learn. The misconception of Unit test Our firmware is very simple, we don’t need unit testing. We can’t unit test microcontroller code, it’s too close to the hardware. Unit testing will add an unnecessary cost burden to our project. What is the unit testing framework? A unit test framework is just some code/application that makes it easier to run, test the code which we have written, and recorded the results of unit tests. Frameworks used for Unit Test So, now we know that unit testing is a valuable activity for embedded firmware. There are many frameworks that are available for each programming language. We have listed some tools for our C and Embedded platform. You can use anyone for unit testing. Ceedling Embunit MinUnit Criterion LCUT etc. In our next tutorial, we will see the code coverage and its types. You can also read the below tutorials. Embedded Software | Firmware | Linux Devic Driver | RTOS Hi, I am a tech blogger and an Embedded Engineer. I am always eager to learn and explore tech-related concepts. And also, I wanted to share my knowledge with everyone in a more straightforward way with easy practical examples. I strongly believe that learning by doing is more powerful than just learning by reading. I love to do experiments. If you want to help or support me on my journey, consider sharing my articles, or Buy me a Coffee! Thank you for reading my blog! Happy learning! Hits (since 1 July 2022) - 32,637 On-demand pricing that scales with youFirst-class support for all major languages and frameworks using lightweight SDKs and agentsNative integrations with Slack, Github, Jira, Bitbucket and more of your favorite tools When it comes to QA, you're not alone. We work side-by-side with your team to ensure your testing process is smooth, efficient, and impactful. Share — copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt — remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation . No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. When you find yourself (or your company) with more code than anyone could ever test by hand, what can you do? Well, unit testing has always been the perfect solution, as you can run tests that check more data than a person could in a day in a matter of milliseconds. So today I’ll take a look into a few popular C# unit testing frameworks and try them out first hand so you can choose which one best suits your project. Unit tests can be run as often as you want, on as many different kinds of data as you want and with next to no human involvement beyond once the tests are written. Not only that, but using code to test code will often result in you noticing flaws with your program that would have been very difficult to spot from a programmer’s viewpoint. Popular C# unit testing frameworks The unit testing frameworks I’ll be testing are: NUnit XUnit Built-in Visual Studio testing tools All of these unit testing frameworks offer a similar end goal, to help make writing unit tests faster, simpler and easier! But there are still a few key differences between them. Some are more focused towards powerful complex tests, while others rank simplicity and usability as a higher priority. In most versions since 2005, Visual Studio has come with a built in testing framework supported by Microsoft. This framework certainly wins the most points for installation. Though if your copy of Visual Studio doesn’t come with it already included you are going to have to jump though a few hoops to get it going. (We wrote a review of the 2017 version of Visual Studio here.) This framework is the simplest of the three, and uses an easy to understand method attribute structure (much like most testing frameworks) where you are able to add tags such as ‘[TestClass]’ and ‘[TestMethod]’ to your code in order to get testing. Visual Studio even has a UI panel dedicated to visualizing your tests, which can be found under Test -> Windows -> Test Explorer. Now before we dive into trying out this testing framework let’s introduce our example classes that need testing. First we have a Raygun, which we can fire and recharge. The only thing we need to keep track of with our Raygun is it’s ammo, which can run out. We also have a bug, which we can shoot at with our Raygun. But this bug has the ability to dodge our attempts to shoot it. If we shoot at a bug after it has just dodged, we will miss. Though if we hit the bug square on, it’s safe to assume that it will be dead. These two classes are defined as follows: public class Raygun { private int ammo = 3; public void FireAt(Bug bug) { if (HasAmmo()) { if (bug.IsDodging()) { bug.Miss(); } else { bug.Hit(); } ammo--; } } public void Recharge() { ammo = 3; } public bool HasAmmo() { return ammo > 0; } } public class Bug { private bool dodging; private bool dead; public void Dodge() { dodging = true; } public void Hit() { dead = true; } public void Miss() { dodging = false; } public bool IsDodging() { return dodging; } public bool IsDead() { return dead; } } Seems simple enough, but we need to make sure that our Rayguns and bugs behave as we want them to. So then it’s time to write some unit tests! (We wrote about how to write robust unit tests in C# here.) First up let’s try a simple situation where we want to shoot at, and hit, a bug. What we would expect is that afterwards the bug will be dead, and the Raygun will still have a bit of juice left in it. Well, let’s see if we are right: [TestClass] public class Class1 { [TestMethod] public void TryShootBug() { Bug bug = new Bug(); Raygun gun = new Raygun(); gun.FireAt(bug); Assert.IsTrue(bug.IsDead()); Assert.IsTrue(gun.HasAmmo()); } } The two new things you will notice in this snippet of code is the [TestClass] and [TestMethod] tags, which certainly don’t just float around in normal code. These tags are what allow Visual Studio’s built in testing framework to recognize this particular class as a class that contains unit tests, and to treat the method TryShootBug() as a test case, instead of just an ordinary method. Since these tools are built for Visual Studio, running your tests from within Visual Studio is very simple. Just right click on any [TestMethod] tags as shown: And would you look at that, the test passed. Looks like our Raygun can at least hit a stationary bug. Of course this is only showing the bare basics of what Visual Studio’s testing tools can do. Some other very useful tags you will surely be using are the [TestInitialize] and [TestCleanup] tags. These tags allow you to specify code that is run before (initialize) and after (cleanup) every individual test is run. So if you want to reload your Raygun after every encounter like a stylish gunslinger, then this should do the trick: [TestInitialize] public void Initialize() { gun = new Raygun(); } [TestCleanup] public void Cleanup() { gun.Recharge(); } Stylish. While we are still talking about the Visual Studio testing tools I’ll quickly mention the [ExpectedException] tag, which is incredibly useful for when you want to deliberately cause an exception in your test (which you will certainly want to do at some point to make sure your program isn’t accepting data it shouldn’t). Here’s a quick example of how you would write a test that results in an exception: [TestMethod] [ExpectedException(typeof(System.IndexOutOfRangeException))] public void TryMakingHeapsOfGuns() { Raygun[] guns = new Raygun[5]; Bug bug = new Bug(); guns[5].FireAt(bug); } Don’t worry about the index out of bounds exception, thanks to the [ExpectedException] tag the exception will be treated as a success and your test will pass. On the contrary if the exception isn’t thrown then the test will fail. Anyway, it’s about time we moved on to some more testing platforms! Overall the built in Visual Studio testing tools do exactly what they say on the box. They are simple, easy to use and handle all the basic testing functionality you would need. Plus if you’re already working in Visual Studio then they are already ready to use! Next up is arguably the most popular C# testing platform, NUnit NUnit is an incredibly widely used tool for testing, and it serves as an excellent example of the open source unit testing frameworks. It’s a broad and powerful testing solution. In fact it’s what we use here at Raygun for the bulk of our unit testing. NUnit is installed via a NuGet package, which you can search for within Visual Studio. The packages I’ve used for this example are NUnit and NUnit.ConsoleRunner, though you also have the option of installing a GUI-based plugin for Visual Studio. NUnit uses a very similar attribute style system just like the visual studio testing tools, but now we will be referring to a [TestClass] as a [TestFixture], and a [TestMethod] as simply a [Test]. Now let’s go back to our Rayguns and bugs and have a look at another example, but this time using NUnit. This time let’s make sure our dodges and ammo are working properly, so let’s try and shoot a much more mobile bug: [TestFixture] public class NUnitTests { [Test] public void TryShootDodgingBug() { Bug bug = new Bug(); Raygun gun = new Raygun(); bug.Dodge(); gun.FireAt(bug); bug.Dodge(); gun.FireAt(bug); bug.Dodge(); gun.FireAt(bug); Assert.IsFalse(bug.IsDead()); Assert.IsFalse(gun.HasAmmo()); } } Notice the new [TestFixture] and [Test] tags. Now in order to run this test using NUnit, we need to seek the command line (unless of course you’ve chosen to install a GUI based plugin.) First, you must make sure you are in your project’s root directory (e.g. C:\UsersyourUserName\Documents\Visual Studio 2015\ProjectsYourProjectName) and then enter the following command in a new cmd window: packages\NUnit.ConsoleRunner.3.6.0\toolsunit3-console.exe YourProjectName\bin\DebugYourProjectName.dll Assuming everything is set up properly, the NUnit console runner will run all the tests in your project and give you a nice little report on how things went: Looks like our bug sure can dodge and our Raygun can certainly run out of ammo! One feature of NUnit that makes it incredibly useful is the ability to include parameters in your tests! This means that you can write a test case with arguments, then easily run the same test with a range of unique data. This removes the need to write unique test cases for every set of arguments you want to test. Here’s a quick example test case we could use to make sure our Raygun was actually running out of ammo at the right time, in a much smarter way than before: [TestCase(1)] [TestCase(2)] [TestCase(3)] [TestCase(4)] public void FireMultipleTimes(int fireCount) { Bug bug = new Bug(); Raygun gun = new Raygun(); for(int i = 0; i < fireCount; i++) { gun.FireAt(bug); } if (fireCount >= 3) { Assert.IsFalse(gun.HasAmmo()); } else { Assert.IsTrue(gun.HasAmmo()); } } Excellent, with this one test case we were able to make sure a Raygun which has fired two shots still has ammo, while one that has fired three is empty. And thanks to the [TestCase] tag we were easily able to test a whole bunch of other values while we were at it! Overall NUnit is an excellent testing framework, and as you delve deeper into what it can offer, it surely exceeds what Microsoft’s built in testing can offer. Anyway, let’s look at our last testing framework, and our last attempt as shooting bugs with Rayguns! If you like the sound of Facts and Theories, then it’s time to look at XUnit XUnit is an open source testing platform with a larger focus in extensibility and flexibility. XUnit follows a more community minded development structure and focuses on being easy to expand upon. XUnit actually refers to a grouping of frameworks, but we will be focusing on the C# version. Other versions include JUnit, a very well known testing framework for Java. XUnit also uses a more modern and unique style of testing, by doing away with the standard [test] [testfixture] terminology and using new fancy tags like Facts and Theories. NUnit and XUnit are actually quite similar in many ways, as NUnit serves as a base for a lot of the new features XUnit brings forward. Note that XUnit is also installed via a NuGet package much like NUnit, which you can search for within Visual Studio. The packages I’ve used for this example are XUnit and XUnit.ConsoleRunner, though you also have the option of installing a GUI-based plugin for Visual Studio. Much like the [TestCase] tag in NUnit, XUnit has its own solution to providing parameters to a test case. To do so we will be using the new [InLineData] tag and Theories. In general, a test case that has no parameters (so it doesn’t rely on any changing data) is referred to as a Fact in XUnit, meaning that it will always execute the same (so ‘Fact’ suits it pretty well). On the other hand, we have Theories, which refers to a test case that can take data directly from [InLineData] tags or even from an Excel spreadsheet So with all these new fancy keywords in mind, let’s write a test in XUnit that uses a theory to test our bugs dodge ability: [Theory] [InlineData(true, false)] [InlineData(false, true)] public void TestBugDodges(bool didDodge, bool shouldBeDead) { Bug bug = new Bug(); Raygun gun = new Raygun(); if (didDodge) { bug.Dodge(); } gun.FireAt(bug); if (shouldBeDead) { Assert.True(bug.IsDead()); } else { Assert.False(bug.IsDead()); } } This test covers both cases at once, where the bug dodges and survives, or doesn’t dodge and gets hit. Lovely! Now, last step, lets run the XUnit test runner from the command line (note that much like NUnit, XUnit also has a GUI based visual studio plugin available for you to run tests with). First you must make sure you are in your project’s root directory, just like NUnit (e.g. C:\UsersyourUserName\Documents\Visual Studio 2015\ProjectsYourProjectName) and then enter the following command in a new cmd window: packages\xunit.runner.console.2.1.0\tools\xunit.console.exe YourProjectName\bin\DebugYourProjectName.dll Assuming everything is set up properly, the XUnit console runner will run all the tests in your project and let you know how your tests turned out. Looks like our dodging tests passed! Overall XUnit acts as the more contemporary version of NUnit, offering flexible and usable testing with a fresh coat of paint. In conclusion… Regardless of which of the unit testing frameworks you use, you’re going to be getting all the basics. However, there are a few differences between them that I hope I’ve highlighted so you can choose the right one for your project. Whether it’s the convenience of Microsoft’s built in unit testing framework, the solid and well proven status of NUnit, or the modern take on unit testing that XUnit provides, theres always something out there that will give you exactly what you need! Want to add an extra layer of protection for your code? Catch the errors that fall through the cracks with Raygun. Take a free trial here. Complex embedded projects have thousands and frequently tens of thousands lines of code. The majority of that code is entirely software (rather than “firmware”), and software in every industry is typically unit tested. However, in the embedded and firmware industry, unit testing is typically an after-thought or a task that is begun after working on a project for months or even years. Today’s firmware projects require filesystems, BLE and Wi-Fi stacks, specialized data structures (both in-memory and in-flash), and complex algorithms, such as those interpreting accelerometer and gyroscope data. All of these items can be easily unit tested after becoming acquainted with best practices and writing a few tests of your own. In this post, we go into detail on how to properly build abstractions to stub, fake, and mock out implementations of low level embedded software and provide a full real-world example of a unit test using the CppUTest 3.8 unit test framework. This is the second post in our Building Better Firmware series, following the post about Continuous Integration for firmware projects, which is a wonderful pre-cursor to this post. Unit Testing Overview Unit testing is a method of testing software where individual software components are isolated and tested for correctness. Ideally, these unit tests are able to cover most if not all of the code paths, argument bounds, and failure cases of the software under test. Through proper use of unit tests, and especially while using practices from Test Driven Development (TDD)1, the time it takes to stabilize embedded software can decrease dramatically, making individuals and teams more productive and firmware less likely to experience functional bugs, control flow bugs, and even fatal issues, such as memory leaks and (gasp!) bootloops. Life Before Unit Testing Here are a few examples that I’ve experienced in the past that were alleviated by the team doubling down on unit testing the firmware. You find testing on target hardware slow and inconvenient, especially when multiple devices (e.g. a mobile phone) or prior setup (e.g. a factory reset) is required Bugs and regressions occur repeatedly in a single piece of software. Deadlocks, HardFaults and Memory Leaks are the norm and have become accepted (unit testing example for preventing deadlocks included below!). The amount of time spent debugging and testing firmware is 60% or more2. The first instinct when starting a new software module is to write a chunk of code and test on hardware. Life After Unit Testing (Possibly) At a previous company, after scrapping most legacy code and writing new modules with 90%+ code coverage and through the use of TDD, this is what development felt like sometimes. You write a new file, maybe an in-flash log storage module, and it works the first time when flashed on the device (no better feeling). Regressions are caught immediately when runnings tests locally or in CI. Memory leaks are raised as errors in unit tests. Testing a majority of the firmware only takes a minute. The overall codebase has better structure and cleaner boundaries between modules. Disclaimers Unit tests in the embedded space is a controversial topics, so I want to clear set up expectations up front. This post covers how to test embedded software. Testing firmware drivers and hardware is very different and time is best spent writing functional and integration tests that run on target to validate hardware components. As soon as drivers are written and stable, switch to the unit test approaches provided in this post. I do not suggest rewriting all of your code to accommodate unit tests, or writing tests for the current code base, but I heavily suggest writing unit tests for most new modules and heavily suggesting them in code reviews. Integration tests and on-target tests have their place. This infrastructure is a huge time and money investment, and the tests run in minutes and hours. Keep these to a minimum at first to ensure hardware stability, and leave software stability to unit tests. If time allows, then build these types of tests. Like Interrupt? Subscribe to get our latest posts straight to your inbox. Framework-less Unit Tests It is very common to initially write unit tests using one-off .c files. Below is an example of a test that is commonly found in firmware projects or written by the author of a piece of firmware code. #include // In my_sum.c int my_sum(int a, int b) { return a + b; } // In test_my_sum.c int main(int argc, char *argv[]) { assert(2 == my_sum(1, 1)); assert(-2 == my_sum(-1, -1)); assert(0 == my_sum(0, 0)); // ... return(0); } This works for a short period of time, but as a firmware project grows in complexity, lines of code, and number of developers, there are a few things that become a requirement. Running Unit Tests in Continuous Integration Reporting results of the tests, e.g. number of tests failed, runtime duration, etc. Reporting code coverage to give insight into how much of a codebase is unit tested3. Ability for a developer to create a new unit test easily and quickly. The most scalable way to write unit tests in C is using a unit testing framework, such as: CppUTest Unity Google Test Even though CppUTest and Google Test are written in C++, they can be used to test C source code, as long as the C header files includes are wrapped with extern "C". extern "C" { #include "my_sum.h" } Minimal Unit Test Example Let’s come up with a bare bones unit test to instrument our simple my_sum module. NOTE: Our examples use the CppUTest framework. If you want to follow along, check out the Setting Up CppUTest section first. The source code for the my_sum.c module is as follows: #include "my_sum.h" int my_sum(int a, int b) { return (a + b); } A unit test generally contains the following pieces: Setup and Teardown functions, which run before and after each test respectively. Individual tests that test logical components or paths of a module. Many checks, such as LONGS_EQUAL which compares integer values and STRCMP_EQUAL which would compare string values. Our basic unit test is as follows: #include "CppUTest/TestHarness.h" extern "C" { #include "my_sum.h" } TEST_GROUP(TestMySum) { void setup() { // This gets run before every test } void teardown() { // This gets run after every test } }; TEST(TestMySum, Test_MySumBasic) { LONGS_EQUAL(7, my_sum(3, 4)); } Although the example is basic, let’s go over what is happening here. We import my_sum.h inside of the extern "C" {} section so that it is compiled as C instead of C++. We have empty setup() and teardown() functions since the modules we are testing don’t require any initial setup or cleanup routines. We have a single LONGS_EQUAL state, which compares == after the my_sum function is called. We did not include any fakes or stubs, as our module didn’t have any dependencies. If this test passes, we get something like: Running build/sum/sum_tests . OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms) And if the test fails (for example, change 7 to 6): Running build/sum/sum_tests src/test_my_sum.cpp:17: error: Failure in TEST(TestMySum, Test_MySumBasic) expected but was . Errors (1 failures, 1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 1 ms) To build and run this unit test, we give the unit test harness the test name, the list of files to compile into the test binary, and any extra compilation flags necessary. COMPONENT_NAME=sum SRC_FILES = \ $(PROJECT_SRC_DIR)/my_sum.c \ TEST_SRC_FILES = \ $(UNITTEST_SRC_DIR)/test_my_sum.c Here, we have SRC_FILES, which would contain any sources files used by the test, and TEST_SRC_FILES which contains the test files that contain the tests themselves. Unit Testing Best Practices The “Minimal Example” is a contrived example and very rarely will there be a test with no other dependencies. Firmware is naturally coupled with other parts of hardware, and that makes it difficult at first to set up a unit test. For example, a flash storage module may call an analytics_inc() function to record the number of writes, a watchdog_feed() function during a large flash erase operation, and timer_schedule() to help defragment the flash later in the future. If we are testing only the flash key/value store, we do not want to include the analytics, watchdog, and timer source files into our unit test. That brings us to a few best practices to follow, especially when writing unit tests for complex and entangled code. Each TEST() within a unit test file should ideally test a single path or feature of the module. A test called TestEverything is an anti-pattern. Each test should be quick. A few milliseconds is ideal, and one second is the worst case run time. Each unit test should ideally include one real implementation of a module. The rest should be stubbed or fake versions of the modules not under test. Those stubbed and fake versions of modules should be written early, reused, and shared. Which brings us to explaining what are stubs, fakes, and mocks? Stubs, Fakes, and Mocks When starting to write unit tests, it is common to write alternate implementations to modules that make sense for a particular unit test. Since unit tests will be run on the host machine, they won’t have hardware, such as an LED. But if a module within a unit test calls Enable_LED(), we could instead have a virtual LED and a state boolean value saving whether the LED is on or off. These alternate implementations of modules have different types. Let’s explain them. Fakes are a working implementation, but will usually substitute their dependencies with something simpler and easier for a test environment. Example: an in-memory key/value store vs a NOR Flash backed Key/Value store. Stubs are a trivial implementation that returns canned values, generally always returning valid or invalid values. Mocks are an implementation that is controlled by the unit test. They can be pre-programmed with return values, check values of arguments, and help verify that functions are called. After having worked at two software oriented hardware companies with 20+ firmware engineers each, my favorite way to organize the test directory is as follows: ├── header_overrides │ ├── string.h │ └── error_codes.h ├── fakes │ ├── fake_analytics.c │ ├── fake_analytics.h │ ├── fake_kv_store.c │ ├── fake_mutex.c │ └── fake_mutex.h ├── stubs │ ├── stub_analytics.h │ ├── stub_kv_store.h │ └── stub_mutex.h ├── mocks │ ├── mock_analytics.h │ └── mock_kv_store.h ├── src │ ├── AllTests.cpp │ ├── test_kv_store.cpp │ ├── test_littlefs_basic.cpp │ └── test_littlefs_format.cpp └── makefiles ├── Makefile_littlefs_basic.mk ├── Makefile_littlefs_format.mk └── Makefile_settings_file.mk where header_overrides/ - headers that override complex, auto-generated, or target specific headers. src/ - The unit tests themselves. makefiles/ - Makefiles to build the individual unit tests. Stubs These are used when the implementation of specific functions or their return values do not matter to the module under test. They are primarily used to fix the linker’s ld: symbol(s) not found errors. These should generally have only a return statement that always returns true, false, 0, NULL, or whatever makes sense in the context of the module. If there is anything more complex than a return statement, consider implementing a Fake instead. Examples: Hardware or peripheral initialization functions since they have little relevance in testing on the host (x86 machine). A time module which returns the time of day (just return a random time). Mutex stubs when the locking or unlocking isn’t being tested. (shown below) #include "mutex/mutex.h" // Stubs Mutex *mutex_create(void) { return NULL; } void mutex_lock(Mutex *mutex) { return; } void mutex_unlock(Mutex *mutex) { return; } Fakes A Fake is commonly used in firmware where it is impractical to use the real implementation for reasons such as: It requires specific hardware (flash chip, peripherals, LED’s, etc.) Examples: A mutex module which checks at the end of the test that all mutexes were properly unlocked (example provided later in the post). A RAM based NOR flash implementation, which flips bits from 1 to 0 when written to and requires “erasing” (flipping bits back to 1) before writing new data. A RAM based key-value store (example shown below). #include "kv_store.h" #include #include typedef struct { char *key; void *val; uint32_t len; } KvEntry; static KvEntry s_kv_store[256]; bool kv_store_write(const char *key, const void *val, uint32_t len) { // Write key/value into RAM store } bool kv_store_read(const char *key, void *buf, uint32_t buf_len, uint32_t *len_read) { // Read key/value from RAM store into buffer } bool kv_store_delete(const char *key) { // Delete key/value from RAM store } Mocks Mocks are incredibly useful if you want to declare each and every return value of a given function. Using many mocks in a single unit test is also the easiest way to test every single code path of the module under test, as you can force any function to return error codes, NULL values, and invalid pointers. These are the most powerful, provide the programmer the most control, and isolate the module under test the best, but they are also the most cumbersome and verbose to use, as every return value has to be pre-programmed. Common mocking frameworks include: We are not going to cover examples of mocks and how to implement them (the topic is big enough for another post), but some pseudo code is shown below to give an understanding: Learn more about mocks in our separate post, Unit Testing with Mocks. TEST(TestKvStore, Test_InitMutexCreated) { // On the next call to `my_malloc`, return the value `NULL`. MOCK_my_malloc.return_value = NULL; // This calls `my_malloc` in its implementation. void *buf = allocate_buffer(); // Ensure that `my_malloc` was called once and only once. LONGS_EQUAL(1, MOCK_my_malloc.call_count); // Ensure that the buffer returned was indeed `NULL` since `my_malloc` returned `NULL`. POINTERS_EQUAL(NULL, buf); } Examples: A malloc implementation that can be pre-programmed with return values (return real buffers vs NULL). A mock flash driver which returns error codes and forces different paths in a higher level module. A Bluetooth socket implementation which is fed artfully crafted packed data to instrument protocols. Later in this post, we will go over how to set up CppUTest to run these examples by downloading the example code, as well as give some short instructions to how to set up your own project to run unit tests. For now, the concepts are more important than the framework and process used to unit test firmware code. Real World Unit Test Example Let’s come up with a more complicated example which more accurately mirrors what a developer on a firmware team would experience. This example uses a stub, a fake, setup() and teardown() functions, and it also compiles littlefs in its entirety, a filesystem by ARM designed for microcontrollers4. Overview We are tasked with writing a Key/Value storage module in a firmware project. The requirements are as follows: The module should have the ability to read, write, and delete key/value pairs. The backing data should be stored in littlefs. The number of times a key/value pair is read, written, or deleted is counted using an analytics.c module with a function call to analytics_increment. This might be used to track roughly how often the flash chip is written to. The module should be locked by a mutex so that only one consumer can be writing, reading, or deleting from the /kv directory of littlefs. In an ideal world, and in our realistic one as well, it is possible for us to write this entire module and test it without actually using real hardware. Let’s get started! Basic Implementation of Key/Value Store Below is our first attempt at kv_store.c which is the skeleton of our file. #include "kv_store.h" #include #include "lfs.h" void kv_store_init(lfs_t *lfs) { } bool kv_store_write(const char *key, const void *val, uint32_t len) { return true; } bool kv_store_read(const char *key, void *buf, uint32_t buf_len, uint32_t *len_read) { return true; } bool kv_store_delete(const char *key) { return true; } Believe it or not, we are ready to create a unit test to test that things are working. It’s usually easier to write a unit test earlier rather than later since the number of dependencies can grow out of hand quickly. Below is a unit test which we mainly create to test compilation and the harness. #include "CppUTest/TestHarness.h" extern "C" { #include "kv_store/kv_store.h" } TEST_GROUP(TestKvStore) { void setup() { } void teardown() { } }; TEST(TestKvStore, Test_SimpleKvStore) { // Just make sure that our file is hooked up LONGS_EQUAL(true, kv_store_read(NULL, NULL, 0, NULL)); } Let’s run the test and see what prints! $ cd complex/tests $ make compiling kv_store.c Building archive build/kv_store/lib/libkv_store.a Linking build/kv_store/kv_store_tests Running build/kv_store/kv_store_tests . OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 1 ms) Looks like our test passes and we are ready to move onto a (more) realistic test. Add littlefs Implementation Our requirement was that our kv_store implementation must use littlefs to store its data. At first, the task seems daunting! How are we supposed to write to a filesystem that doesn’t exist on our host machine? Also, a filesystem is a complicated piece of software! Thankfully, littlefs includes an emulated version of its filesystem which runs directly on a PC. These source files are under littlefs/emubd, and we can add them to our unit test to make a fully functional littlefs filesystem. In this example, we can imagine that the emubd portion of littlefs is a fake. The strategy we use to store various key/value pairs is that each key will be a new filename under the /kv directory, and the value will be written as the file data. Let’s try writing the source code! #define SETTINGS_DIR "/kv" static char s_fname[256]; static lfs_file_t s_file; static lfs_t *s_lfs_ptr; static const char *prv_prefix_fname(const char *key) { snprintf(s_fname, sizeof(s_fname), "%s/%s", SETTINGS_DIR, key); return s_fname; } void kv_store_init(lfs_t *lfs) { s_lfs_ptr = lfs; lfs_mkdir(s_lfs_ptr, "/kv"); } bool kv_store_write(const char *key, const void *val, uint32_t len) { lfs_file_open(s_lfs_ptr, &s_file, prv_prefix_fname(key), LFS_O_WRONLY | LFS_O_CREAT); uint32_t rv = lfs_file_write(s_lfs_ptr, &s_file, val, len); lfs_file_close(s_lfs_ptr, &s_file); return (rv == len); } bool kv_store_read(const char *key, void *buf, uint32_t buf_len, uint32_t *len_read) { int rv = lfs_file_open(s_lfs_ptr, &s_file, prv_prefix_fname(key), LFS_O_RDONLY); if (rv < 0) { return false; } uint32_t len = lfs_file_size(s_lfs_ptr, &s_file); if (buf_len < len) { return false; } len = lfs_file_read(s_lfs_ptr, &s_file, buf, buf_len); lfs_file_close(s_lfs_ptr, &s_file); *len_read = len; return len; } bool kv_store_delete(const char *key) { lfs_remove(s_lfs_ptr, prv_prefix_fname(key)); return true; } This is a reasonable start for our module. It could use more error checking, but the basics are there. Let’s test things out. One thing to note is that we’ll have to add the source files for littlefs, so we add those in our Makefile as shown below. If we try to run the unit test without adding the source files, we run into linker errors telling us that symbols are missing. $ make Linking build/kv_store/kv_store_tests Undefined symbols for architecture x86_64: "_lfs_file_close", referenced from: _kv_store_write in libkv_store.a(kv_store.o) _kv_store_read in libkv_store.a(kv_store.o) "_lfs_file_open", referenced from: _kv_store_write in libkv_store.a(kv_store.o) _kv_store_read in libkv_store.a(kv_store.o) ... We can simply add these source files to our compilation and then all should be well. If these new files had dependencies of their own, we’d have to fix those linker errors as well. COMPONENT_NAME=kv_store SRC_FILES = \ $(PROJECT_SRC_DIR)/littlefs/lfs.c \ $(PROJECT_SRC_DIR)/littlefs/lfs_util.c \ $(PROJECT_SRC_DIR)/littlefs/emubd/lfs_emubd.c \ $(PROJECT_SRC_DIR)/kv_store/kv_store.c \ TEST_SRC_FILES = \ $(UNITTEST_SRC_DIR)/test_kv_store.c include $(CPPUTEST_MAKFILE_INFRA) Add littlefs Initialization Just because we got the files to compile in our unit test does not mean littlefs will magically work. We need to initialize the filesystem and set up and tear it down before and after each test respectively. To learn how to do this, we can go to the existing littlefs tests directory and take inspiration from the template and a basic file test, both linked below. This ultimately results in the following changes necessary for the unit test file. extern "C" { ... #include "lfs.h" #include "emubd/lfs_emubd.h" // Contains lfs, cfg, variables with default configuration. #include "defs/lfs_default_config.h" } TEST_GROUP(TestKvStore) { void setup() { lfs_emubd_create(&cfg, "blocks"); lfs_format(&lfs, &cfg); lfs_mount(&lfs, &cfg); kv_store_init(&lfs); } void teardown() { lfs_emubd_destroy(&cfg); lfs_unmount(&lfs); } }; The unit test will now, at the start of every test, create a directory called blocks/, format and mount the filesystem there, and initialize the key/value store, and at the end of the test, destroy and unmount the filesystem so the next test starts with a clean environment. Since we now have a filesystem backing our key/value store, we can write a simple test! TEST(TestKvStore, Test_SimpleKvStore) { bool success; const char *key = "hello"; const char *val = "world"; success = kv_store_write(key, (void *)val, sizeof(val)); CHECK(success); char buf[16]; uint32_t read_len; success = kv_store_read(key, buf, sizeof(buf), &read_len); CHECK(success); STRCMP_EQUAL(val, buf); // Buffer length too short. Should return false. success = kv_store_read(key, buf, 0, &read_len); CHECK_FALSE(success); } This test writes a key “hello” with the value “world”, reads the value stored at key “hello” into a buffer, and compares it against the expected value “world”. We also check the failure case of kv_store_read by passing in a buffer that is too small. It passes! This means our littlefs was set up correctly, and that our initial logic in kv_store.c was (mostly) correct. Add Analytics Our next requirement was to add analytics tracking how many times key/value pairs were written, read, and deleted. We can do this by simply calling a function analytics_inc which will increment the count of the given key by one. The additions to our source code are shown below: ... #include "analytics/analytics.h" bool kv_store_write(const char *key, const void *val, uint32_t len) { ... analytics_inc(kSettingsFileWrite); return (rv == len); } bool kv_store_read(const char *key, void *buf, uint32_t buf_len, uint32_t *len_read) { ... analytics_inc(kSettingsFileRead); return len; } bool kv_store_delete(const char *key) { ... analytics_inc(kSettingsFileDelete); return true; } If we run the test now, as usual, we will receive linker errors. Linking build/kv_store/kv_store_tests Undefined symbols for architecture x86_64: "_analytics_inc", referenced from: _kv_store_write in libkv_store.a(kv_store.o) _kv_store_read in libkv_store.a(kv_store.o) _kv_store_delete in libkv_store.a(kv_store.o) Since this isn’t a core functionality of the kv_store module, and it’s likely something we don’t need to verify, we are going to provide a stub for this function, rather than use the real implementation or a fake. We can do that by creating a header called stub_analytics.h // Analytics Stub #include "analytics/analytics.h" void analytics_inc(eAnalyticsKey key) { return; } which we include in our unit test as shown below. extern "C" { ... #include "stubs/stub_analytics.h" } Add Mutex Locking Almost done! We’ve also been advised to add locking around our filesystem calls to ensure that only one client can read and write to the /kv directory at one time. In our implementation, we add mutex_lock() and a mutex_unlock() at the start of the function and just before the analytics_inc calls respectively. #include "mutex/mutex.h" static Mutex *s_mutex; void kv_store_init(lfs_t *lfs) { ... s_mutex = mutex_create(); } bool kv_store_write(const char *key, const void *val, uint32_t len) { mutex_lock(s_mutex); // New ... mutex_unlock(s_mutex); // New analytics_inc(kSettingsFileWrite); return (rv == len); } bool kv_store_read(const char *key, void *buf, uint32_t buf_len, uint32_t *len_read) { mutex_lock(s_mutex); // New int rv = lfs_file_open(s_lfs_ptr, &s_file, prv_prefix_fname(key), LFS_O_RDONLY); if (rv < 0) { return false; } uint32_t len = lfs_file_size(s_lfs_ptr, &s_file); if (buf_len < len) { return false; } len = lfs_file_read(s_lfs_ptr, &s_file, buf, buf_len); lfs_file_close(s_lfs_ptr, &s_file); *len_read = len; mutex_unlock(s_mutex); // New analytics_inc(kSettingsFileRead); return true; } bool kv_store_delete(const char *key) { mutex_lock(s_mutex); // New ... mutex_unlock(s_mutex); // New analytics_inc(kSettingsFileDelete); return true; } If we run our test now, we’ll receive linker errors for the missing mutex_* symbols. Since mutexes are rather important to this module, and we wouldn’t want to forget to unlock a mutex, we are going to try writing a fake mutex implementation instead of a stub. Fake Mutex Implementation The reason we chose to write a fake implementation for the mutex module is that we want to ensure that equal number of lock and unlock calls are made so that there are no bugs when we actually use the kv_store in a real environment. This may seem difficult, but it’s rather easy. To create a fake, we create two files, fake_mutex.h, and fake_mutex.c. The reason for both the .h and .c files is because the fake implementation defines new functions that are only relevant to using the fake in a unit test. Here is most of the source code for fake_mutex.c. #define NUM_MUTEXES 256 typedef struct Mutex { uint8_t lock_count; } Mutex; static Mutex s_mutexes[NUM_MUTEXES]; static uint32_t s_mutex_index; // Fake Helpers void fake_mutex_init(void) { memset(s_mutexes, 0, sizeof(s_mutexes)); } bool fake_mutex_all_unlocked(void) { for (int i = 0; i < NUM_MUTEXES; i++) { if (s_mutexes[i].lock_count > 0) { return false; } } return true; } // Implementation Mutex *mutex_create(void) { assert(s_mutex_index < NUM_MUTEXES); return &s_mutexes[s_mutex_index++]; } void mutex_lock(Mutex *mutex) { mutex->lock_count++; } void mutex_unlock(Mutex *mutex) { mutex->lock_count--; } Let’s go over what the fake is doing. We allocate a large number (256) of mutex slots. Since we are running our test on a host, large allocations are fine. We have gigabytes of RAM, unlike the limited embedded counterparts. We define a new typedef struct Mutex type which only stores the lock_count. If the type Mutex was properly hidden within a .c file in the real implementation, this should work. fake_mutex_init resets the state of this module. This should be called in the setup() function of every unit test using this module. Otherwise, the state will be carried over between tests, which isn’t desired. fake_mutex_all_unlocked ensures that all mutexes are unlocked when called. We can call this manually or at the end of every test in the teardown() function. mutex_create allocates a slot for a new mutex in our array and returns this pointer to the client. Since the client only uses the Mutex * as an opaque type, it shouldn’t matter that it’s a fake Mutex. mutex_lock and mutex_unlock increment and decrement the lock count respectively. Now we can use this fake in our unit test. After adding the source file to our compilation scripts, we make the follow changes to the unit test. TEST_GROUP(TestKvStore) { void setup() { fake_mutex_init(); ... } void teardown() { ... CHECK(fake_mutex_all_unlocked()); } }; If we run our code above with the naive top/bottom mutex additions, we realize quickly that we have a mistake. Our fake_mutex_all_unlocked check failed! complex/tests/src/test_kv_store.cpp:42: error: Failure in TEST(TestKvStore, Test_SimpleKvStore) complex/tests/src/test_kv_store.cpp:38: error: CHECK(fake_mutex_all_unlocked()) failed . Errors (1 failures, 1 tests, 1 ran, 5 checks, 0 ignored, 0 filtered out, 8 ms) Looking back, hopefully the issue is obvious. We forgot to unlock on our failure cases within kv_store_read. The follow change is necessary: if (rv < 0) { mutex_unlock(s_mutex); // ADD return false; } uint32_t len = lfs_file_size(s_lfs_ptr, &s_file); if (buf_len < len) { mutex_unlock(s_mutex); // ADD return false; } Thankfully we wrote and used our fake mutex implementation, as deadlocks are the worst to debug! If you do find that deadlocks are a constant issue, do check out Memfault. It will help you track them down easily. Setting Up CppUTest CppUTest is one of many C/C++ unit test frameworks, and the reason it was chosen is because of my familiarity with it and that it doesn’t have any dependencies other than Make. No matter what anyone says, the framework you use does not matter. As long as the framework has the minimum features listed above, it is as good as any. Initial Setup We first need to install a pre-compiled version of CppUTest so we can easily run tests without needing to compile the binary ourselves from source before every test run. The easiest way to do this is to use your system’s package manager. On macOS, CppUTest can be installed using brew: On Ubuntu, it can be installed using apt: $ sudo apt install cpputest Project CppUTest Harness Since it is a decent amount of boilerplate due to the CppUTest harness setup required, this example will start with a clone of the example repository and then go over the components within it, briefly covering the CppUTest Makefiles. Feel free to use the code in any way you like, and even copy it into your project. It should build quite easily once some paths are patched up. $ git clone $ cd interrput/examples/unit-testing/minimal/tests # macOS $ make # Ubuntu $ make CPPUTEST_HOME=/usr TARGET_PLATFORM=x86_64-linux-gnu compiling test_my_sum.cpp compiling AllTests.cpp ... Linking build/sum/sum_tests Running build/sum/sum_tests . OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 1 ms) To set CPPUTEST_HOME and TARGET_PLATFORM for your platform, edit the first few lines of MarkefileWorkerOverrides.mk. CPPUTEST_HOME ?= /usr/local/Cellar/cpputest/3.8 TARGET_PLATFORM ?= Tips & Tricks Debugging a Unit Test Most unit test frameworks will generate separate binaries for each .cpp unit test file written so that you can load them in a debugger (lldb or gdb). $ lldb build/sum/sum_tests (lldb) target create "build/sum/sum_tests" Current executable set to 'build/sum/sum_tests' (x86_64). (lldb) run Process 257894 launched: 'build/sum/sum_tests' (x86_64) . OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms) Process 257894 exited with status = 0 (0x00000000) Code Coverage One of the wonderful parts about unit testing is that you can generate a code coverage report. This shows which paths were covered in a given set of unit tests, so you can be sure that the piece of code was tested in some capacity. Note that code coverage doesn’t measure the different behaviors a code path could take, but only that a particular code path was taken. To generate a coverage report for our minimal example, let’s first install lcov. # macOS $ brew install lcov # Linux $ sudo apt install lcov Next, we’ll run our unit tests while testing for coverage. $ make lcov make -f minimal/tests/makefiles/Makefile_sum.mk make[1]: Entering directory 'minimal/tests' Running build/sum/sum_tests . OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms) make[1]: Leaving directory 'minimal/tests' lcov --base-directory . --directory . -c -o build/lcov.info --exclude "*cpputest/*" --exclude "*tests/*" ... Overall coverage rate: lines......: 100.0% (2 of 2 lines) functions..: 100.0% (1 of 1 function) You can see the very end reports a simple coverage report in the terminal, but a more detailed report can be found in the HTML website that was generated. We can open it from the terminal: # macOS $ open build/test_coverage/index.html # Linux $ firefox build/test_coverage/index.html Below is the coverage report for our minimal example. It’s quite basic because there isn’t much code being tested. Below is a more realistic report from the Memfault Public SDK5. Address Sanitizing To raise use-after-free and buffer overflow errors in unit tests, use the compiler option -fsanitize=address when compiling unit tests. You can find out more about the Address Sanitizer by reading the documentation6. Common Issues with C/C++ Unit Tests Writing unit tests in C isn’t as simple as writing tests in some languages. Here are some common errors and mistakes that everyone runs into and possible solutions. Linking build/kv_store/kv_store_tests Undefined symbols for architecture x86_64: "_analytics_inc", referenced from: _kv_store_write in libkv_store.a(kv_store.o) _kv_store_read in libkv_store.a(kv_store.o) _kv_store_delete in libkv_store.a(kv_store.o) ld: symbol(s) not found for architecture x86_64 This error is generated by the linker to tell the user that there are undefined symbols. In the example above, the function analytics_inc is called from three different functions, but isn’t defined anywhere. The possible solutions to the issue are: Create a fake, stub, or mock file which implements this function and add it either to the headers or compile it into the library Define the function within the unit test file itself, probably at the top of the file. If it is a C function being called by C code, place it within the extern "C" {} section. As a last resort, one can compile out the function calls and implementations using a define for unit tests. e.g. #if !INSIDE_UNITTESTS Linking build/kv_store/kv_store_tests duplicate symbol '_mutex_create' in: build/kv_store/objs/complex/tests/src/test_kv_store.o build/kv_store/lib/libkv_store.a(fake_mutex.o) ld: 1 duplicate symbol for architecture x86_64 This error is generated by the linker when more than one implementation of a function is found and used. This is usually because more than one of the following were included in the unit test: real implementation, fake, stub, or mock. In the example above, I had included a fake_mutex.c file and included the stub_mutex.h header, which caused a duplicate mutex_create symbol. The solution would be to remove one or the other. State carrying over between tests If there is a fake or module which contains static or global state, and it is being used across multiple tests in a single file, then that state ideally should be cleared out. This is usually done by: Defining a fake__reset if the module is a fake. Defining a _deinit if the module is a real one that is used in production code. Make sure to compile this function out of the production code by using #if !INSIDE_UNITTESTS, or ensuring the linker removes it from the final binary. Code space is precious! Final Thoughts Unit testing was something that a co-worker of mine suggested to me 4 years ago when writing a complicated raw flash storage to filesystem migration for our firmware. After having spent a month doing cycling between the 1. Write Code, 2. Setup hardware, 3. Test, which took 10 minutes each iteration, I invested 2 days writing a unit test and was able to shrink the test cycle down to 2 seconds. The rest of the project took two days. Everyone has a moment where unit testing finally clicks for them and this was mine. I hope this post has been useful and that it has inspired you to consider writing a unit test for your next new embedded software module. Want to keep reading? Check out our next post about unit testing, Unit Testing with Mocks. You can find the examples shown in this post here. Like Interrupt? Subscribe to get our latest posts straight to your inbox.