Unity – Getting Started

Welcome

Congratulations. You're now the proud owner of your very own pile of bits! What are you going to do with all these ones and zeros? This document should be able to help you decide just that.

Unity is a unit test framework. The goal has been to keep it small and functional. The core Unity test framework is three files: a single C file and a couple header files. These team up to provide functions and macros to make testing easier.

Unity was designed to be cross-platform. It works hard to stick with C standards while still providing support for the many embedded C compilers that bend the rules. Unity has been used with many compilers, including GCC, IAR, Clang, Green Hills, Microchip, and MS Visual Studio. It's not much work to get it to work with a new target.

Overview of the Documents

Unity Assertions Reference

This document will guide you through all the assertion options provided by Unity. This is going to be your unit testing bread and butter. You'll spend more time with assertions than any other part of Unity.

Unity Assertions Cheat Sheet

This document contains an abridged summary of the assertions described in the previous document. It's perfect for printing and referencing while you familiarize yourself with Unity's options.

Unity Configuration Guide

This document is the one to reference when you are going to use Unity with a new target or compiler. It'll guide you through the configuration options and will help you customize your testing experience to meet your needs.

Unity Helper Scripts

This document describes the helper scripts that are available for simplifying your testing workflow. It describes the collection of optional Ruby scripts included in the auto directory of your Unity installation. Neither Ruby nor these scripts are necessary for using Unity. They are provided as a convenience for those who wish to use them.

Unity License

What's an open source project without a license file? This brief document describes the terms you're agreeing to when you use this software. Basically, we want it to be useful to you in whatever context you want to use it, but please don't blame us if you run into problems.

Overview of the Folders

If you have obtained Unity through Github or something similar, you might be surprised by just how much stuff you suddenly have staring you in the face. Don't worry, Unity itself is very small. The rest of it is just there to make your life easier. You can ignore it or use it at your convenience. Here's an overview of everything in the project.

- `src` — This is the code you care about! This folder contains a C file and two header files. These three files *are* Unity.
- `docs` — You're reading this document, so it's possible you have found your way into this folder already. This is where all the handy documentation can be found.
- `examples` — This contains a few examples of using Unity.
- `extras` — These are optional add-ons to Unity that are not part of the core project. If you've reached us through James Grenning's book, you're going to want to look here.
- `test` — This is how Unity and its scripts are all tested. If you're just using Unity, you'll likely never need to go in here. If you are the lucky team member who gets to port Unity to a new toolchain, this is a good place to verify everything is configured properly.
- `auto` — Here you will find helpful Ruby scripts for simplifying your test workflow. They are purely optional and are not required to make use of Unity.

How to Create A Test File

Test files are C files. Most often you will create a single test file for each C module that you want to test. The test file should include unity.h and the header for your C module to be tested.

Next, a test file will include a `setUp()` and `tearDown()` function. The setUp function can contain anything you would like to run before each test. The tearDown function can contain anything you would like to run after each test. Both functions accept no arguments and return nothing. You may leave either or both of these blank if you have no need for them. If you're using a compiler that is configured to make these functions optional, you may leave them off completely. Not sure? Give it a try. If you compiler complains that it can't find setUp or tearDown when it links, you'll know you need to at least include an empty function for these.

The majority of the file will be a series of test functions. Test functions follow the convention of starting with the word "test" or "spec". You don't HAVE to name them this way, but it makes it clear what functions are tests for other developers. Test functions take no arguments and return nothing. All test accounting is handled internally in Unity.

Finally, at the bottom of your test file, you will write a `main()` function. This function will call `UNITY_BEGIN()`, then `RUN_TEST` for each test, and finally `UNITY_END()`. This is what will actually trigger each of those test functions to run, so it is important that each function gets its own `RUN_TEST` call.

Remembering to add each test to the main function can get to be tedious. If you enjoy using helper scripts in your build process, you might consider making use of our handy generate_test_runner.rb script. This will create the main function and all the calls for you, assuming that you have followed the suggested naming conventions. In this case, there is no need for you to include the main function in your test file at all.

When you're done, your test file will look something like this:

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

void setUp(void) {
    // set stuff up here
}

void tearDown(void) {
    // clean stuff up here
}

void test_function_should_doBlahAndBlah(void) {
    //test stuff
}

void test_function_should_doAlsoDoBlah(void) {
    //more test stuff
}

int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_function_should_doBlahAndBlah);
    RUN_TEST(test_function_should_doAlsoDoBlah);
    return UNITY_END();
}
```

It's possible that you will require more customization than this, eventually. For that sort of thing, you're going to want to look at the configuration guide. This should be enough to get you going, though.

How to Build and Run A Test File

This is the single biggest challenge to picking up a new unit testing framework, at least in a language like C or C++. These languages are REALLY good at getting you "close to the metal"

(why is the phrase metal? Wouldn't it be more accurate to say "close to the silicon"?). While this feature is usually a good thing, it can make testing more challenging.

You have two really good options for toolchains. Depending on where you're coming from, it might surprise you that neither of these options is running the unit tests on your hardware. There are many reasons for this, but here's a short version:

- On hardware, you have too many constraints (processing power, memory, etc)
- On hardware, you don't have complete control over all registers.
- On hardware, unit testing is more challenging
- Unit testing isn't System testing. Keep them separate.

Instead of running your tests on your actual hardware, most developers choose to develop them as native applications (using gcc or MSVC for example) or as applications running on a simulator. Either is a good option. Native apps have the advantages of being faster and easier to set up. Simulator apps have the advantage of working with the same compiler as your target application. The options for configuring these are discussed in the configuration guide.

To get either to work, you might need to make a few changes to the file containing your register set (discussed later).

In either case, a test is built by linking unity, the test file, and the C file(s) being tested. These files create an executable which can be run as the test set for that module. Then, this process is repeated for the next test file. This flexibility of separating tests into individual executables allows us to much more thoroughly unit test our system and it keeps all the test code out of our final release!