(a simple Unit Test Framework for Embedded C)



Unity is a unit test framework. Our goal has been to keep is small and functional. The core Unity test framework is a single C and header file pair, which provide functions and macros to make testing easier. Most of it is a variety of assertions which are meant to be placed in tests to verify that variables and return values contain the information that you believe they should. There are some additional methods for general test flow control.

We've developed Unity to be fairly cross-platform. It uses ANSI C for the library itself, and beautiful cross-platform Ruby for all the optional add-on scripts. We've personally used Unity with GCC, IAR's Embedded C Compiler, Clang, and MS Visual Studio. It shouldn't be too much work to get it to work with something else.

When you download Unity, you're going to get some other goodies as well. Let's look at the root directory for a hint as to what those goodies are:

- test – These are tests using Unity which actually test unity itself. How cool is that?
- src – This is where Unity, and some example helpers to expand Unity, live
- examples – This has examples of how to use Unity and it's scripts.
- docs – This has our wonderfully entertaining documentation (like this file)
- build – Just ignore this. Temporary build files get thrown here.
- auto – This contains a collection of Ruby scripts which are going to make using Unity less painful
- targets – This is full of yaml configuration files. They are primarily here for unity's self tests when running rake (see config[] options) but are also a good reference for options used.

You're also going to see a makefile and a rakefile (plus a rakefile_helper... which strangely enough just helps the rakefile).

The makefile is a simple makefile which can be used to get the Unity tests going... it's also a good example of a simple way of assembling your tests. It's currently written assuming you are using GNU Make, but is simple enough that you could tweak it to use with something else.

The rakefile does the same thing, but using Rake. If you have Ruby and Rake installed you can use Rake instead of Make. It's going to provide you with extra goodies like test summaries and the ability to automagically discover your test functions (so you don't have to remember to call each one by hand).

# How To Use Unity

We often run our Unit tests in a simulator. If it's not possible or inconvenient for you, you may also be able to build them into an executable locally and run it. The biggest thing you are missing out on if you do this is the ability to have your code and/or tests directly write to any arbitrary location in memory. This is extremely useful when you want to read or write “registers”.

The Unit tests get built in little chunks. Each module you want to test is built with it's corresponding test module, a test runner, and whatever other supporting modules are needed. This test is then run before moving on to the next module. Unit Tests DO NOT end up in your final release (or even debug) executable, because they are built separately.

If you are using the scripts in the auto directory, you get some extra niceties. First, you don't have to write those TestRunner files. These are automatically generated for you. Second, you don't have to search the results, a report will be generated for you.

# Unity Test API

## *Running Tests*

| | |
|---|---|
| `RUN_TEST(func)` | Each Test is run within the macro `RUN_TEST`. This macro performs necessary setup before the test is called and handles cleanup and result tabulation afterwards. |

## *Ignoring Tests*

There are times when a test is incomplete or not valid for some reason. At these times, `TEST_IGNORE` can be called. Control will immediately be returned to the caller of the test, and no failures will be returned.

| | |
|---|---|
| `TEST_IGNORE()` | Ignore this test and return immediately |
| `TEST_IGNORE_MESSAGE (message)` | Ignore this test and return immediately. Output a message stating why the test was ignored. |

### *Aborting Tests*

There are times when a test will contain an infinite loop on error conditions, or there may be reason to escape from the test early without executing the rest of the test. A pair of macros support this functionality in Unity. The first (`TEST_PROTECT`) sets up the feature, and handles emergency abort cases. `TEST_ABORT` can then be used at any time within the tests to return to the last `TEST_PROTECT` call.

| | |
|---|---|
| `TEST_PROTECT()` | Setup and Catch macro |
| `TEST_ABORT()` | Abort Test macro |

*Example*:

```
main()
{
    if (TEST_PROTECT() == 0)
    {
        MyTest();
    }
}
```

If MyTest calls `TEST_ABORT`, program control will immediately return to `TEST_PROTECT` with a non-zero return value.

## Unity Assertion Summary

### *Basic Validity Tests*

| | |
|---|---|
| `TEST_ASSERT_TRUE (condition)` | Evaluates whatever code is in condition and fails if it evaluates to false |
| `TEST_ASSERT_FALSE (condition)` | Evaluates whatever code is in condition and fails if it evaluates to true |
| `TEST_ASSERT (condition)` | Another way of calling `TEST_ASSERT_TRUE` |
| `TEST_ASSERT_UNLESS (condition)` | Another way of calling `TEST_ASSERT_FALSE` |
| `TEST_FAIL (message)` | This test is automatically marked as a failure. The message is output stating why. |

### *Numerical Assertions: Integers*

All the TEST_ASSERT_EQUAL macros come in a few flavors. In addition to the basic ones listed, you can append _MESSAGE to add an additional message string argument (the custom message will be placed after the standard output) or add _ARRAY to work with an array of those elements. The number of elements to check is passed in as the third argument. You can even do both.

| | |
|---|---|
| `TEST_ASSERT_EQUAL (expected, actual)` | Another way of calling `TEST_ASSERT_EQUAL_INT` |
| `TEST_ASSERT_EQUAL_INT (expected, actual)` | Compare two integers for equality and display errors as signed integers. If the ints passed are smaller, they will be cast to full size, so you can just use this most of the time instead of using a specific item like the next three. |
| `TEST_ASSERT_EQUAL_INT8 (expected, actual)` | Compare two 8-bit integers for equality and display errors as signed integers. |
| `TEST_ASSERT_EQUAL_INT16 (expected, actual)` | Compare two 16-bit integers for equality and display errors as signed integers. |
| `TEST_ASSERT_EQUAL_INT32 (expected, actual)` | Compare two 32-bit integers for equality and display errors as signed integers. |
| `TEST_ASSERT_EQUAL_INT64 (expected, actual)` | Compare two 64-bit integers for equality and display errors as signed integers. (if enabled) |
| `TEST_ASSERT_EQUAL_UINT (expected, actual)` | Compare two integers for equality and display errors as unsigned integers. Like _INT above, you can use this instead of the specific versions in most cases. |
| `TEST_ASSERT_EQUAL_UINT8 (expected, actual)` | Compare two 8-bit integers for equality and display errors as unsigned integers. |
| `TEST_ASSERT_EQUAL_UINT16 (expected, actual)` | Compare two 16-bit integers for equality and display errors as unsigned integers. |
| `TEST_ASSERT_EQUAL_UINT32 (expected, actual)` | Compare two 32-bit integers for equality and display errors as unsigned integers. |
| `TEST_ASSERT_EQUAL_UINT64 (expected, actual)` | Compare two 64-bit integers for equality and display errors as unsigned integers. (if enabled) |
| `TEST_ASSERT_EQUAL_HEX8 (expected, actual)` | Compare two integers for equality and display errors as an 8-bit hex value |
| `TEST_ASSERT_EQUAL_HEX16 (expected, actual)` | Compare two integers for equality and display errors as an 16-bit hex value |
| `TEST_ASSERT_EQUAL_HEX32 (expected, actual)` | Compare two integers for equality and display errors as an 32-bit hex value |
| `TEST_ASSERT_EQUAL_HEX64 (expected, actual)` | Compare two integers for equality and display errors as an 64-bit hex value (if enabled) |
| `TEST_ASSERT_EQUAL_HEX (expected, actual)` | Another way of calling `TEST_ASSERT_EQUAL_HEX32` |

## *Numerical Assertions: Integer Ranges*

| | |
|---|---|
| `TEST_ASSERT_INT_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. Failures are displayed as signed integers. |
| `TEST_ASSERT_UINT_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. Failures are displayed as signed integers. |
| `TEST_ASSERT_HEX8_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. Failures are displayed as 2 nibble hex. |
| `TEST_ASSERT_HEX16_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. Failures are displayed as 4 nibble hex. |
| `TEST_ASSERT_HEX32_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. Failures are displayed as 8 nibble hex. |
| `TEST_ASSERT_HEX64_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. Failures are displayed as 16 nibble hex. (if enabled) |

## *Numerical Assertions: Bitwise*

| | |
|---|---|
| `TEST_ASSERT_BITS (mask, expected, actual)` | Use an integer mask to specify which bits should be compared between two other integers. High bits in the mask are compared, low bits ignored. |
| `TEST_ASSERT_BITS_HIGH (mask, actual)` | Use an integer mask to specify which bits should be inspected to determine if they are all set high. High bits in the mask are compared, low bits ignored. |
| `TEST_ASSERT_BITS_LOW (mask, actual)` | Use an integer mask to specify which bits should be inspected to determine if they are all set low. High bits in the mask are compared, low bits ignored. |
| `TEST_ASSERT_BIT_HIGH (bit, actual)` | Test a single bit and verify that it is high. The bit is specified 0-31 for a 32-bit integer. |
| `TEST_ASSERT_BIT_LOW (bit, actual)` | Test a single bit and verify that it is low. The bit is specified 0-31 for a 32-bit integer. |

### *Numerical Assertions: Floats*

| | |
|---|---|
| `TEST_ASSERT_FLOAT_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. |
| `TEST_ASSERT_EQUAL_FLOAT (expected, actual)` | Asserts that the actual value is within a couple of significant bits of the expected value. |
| `TEST_ASSERT_EQUAL_FLOAT_ARRAY (expected, actual, num_elements)` | Yes, floats get array handlers too |

### *String Assertions*

These strings, as well as the strings you specify in the "message" parameter of all the _MESSAGE macros, will do a little bit of work on your strings when they are displayed. It replaces carriage returns and line feeds with the traditional \r and \n. Other non-printable chars are displayed like so \0x01.

| | |
|---|---|
| `TEST_ASSERT_EQUAL_STRING (expected, actual)` | Compare two null-terminate strings. Fail if any character is different or if the lengths are different. |
| `TEST_ASSERT_EQUAL_STRING_MESSAGE (expected, actual, message)` | Compare two null-terminate strings. Fail if any character is different or if the lengths are different. Output a custom message on failure. |

### *Pointer Assertions*

Most pointer operations can be performed by simply using the integer comparisons above. However, a couple of special cases are added for clarity.

| | |
|---|---|
| `TEST_ASSERT_NULL (pointer)` | Fails if the pointer is not equal to NULL |
| `TEST_ASSERT_NOT_NULL (pointer)` | Fails if the pointer is equal to NULL |
| `TEST_ASSERT_EQUAL_POINTER (expected, actual)` | Verifies that two pointers are the same. This is just like a hex comparison, but it's always the right size int for your system's pointers. |

### *Memory Assertions (for all your other weird types)*

| | |
|---|---|
| `TEST_ASSERT_EQUAL_MEMORY (expected, actual, len)` | Compare two blocks of memory. This is useful for packed structs, buffers, etc... just keep in mind that it's checking everything in that range... so if your struct is unpacked, you might get false failures. |
| `TEST_ASSERT_EQUAL_MEMORY_MESSAGE` `TEST_ASSERT_EQUAL_MEMORY_ARRAY` | Yes, memory compares come in those convenient variations too |

## Helper Scripts

### *generate_test_runner.rb*

This script will allow you to specify any test file name in your project and will automatically create a test runner (which includes "main") to run that test. It searches your test file for void-returning functions starting with "test". It assumes all of these functions are tests and builds up a test suite for you. For example, the following would be tests:

```
void testVerifyThatUnityIsAwesomeAndWillMakeYourLifeEasier(void) {
  ASSERT_TRUE(1);
}

void test_FunctionName_WorksProperlyAndAlwaysReturns8(void) {
  ASSERT_EQUAL(8, FunctionName());
}
```

You can run this script from the command line or make use of it through other Ruby scripts by including the file and then instantiating the class. Let's look at the command line usage:

```
ruby generate_test_runner.rb test_file_being_tested name_of_runner
```

or you can automatically name the runner by just using

```
ruby generate_test_runner.rb test_file_being_tested
```

If you are using Ruby and Rake, there is a much better way to do all this. You can take advantage of some of the extra features of this script, including the ability to push your own header files into your test runners and the ability to get a list of all the header files included by a test (for easy test building). This is demonstrated in the *examples* directory.

### *unity_test_summary.rb*

This script will generate a summary of your test output for you. It tells you how many tests were run, how many were ignore, and how many failed. It also gives you a listing of which tests specifically were ignored and failed. It does this by searching results files that you pass to it. A great example of this is also in the *examples* directory. There are intentional ignored and failing tests in this project in order to demonstrate what these situations look like in a summary report.

## Options

When you're compiling tests with Unity, you can optional include the following #defines to override the default behaviors and customize your experience a little... very likely you'll include them as compiler switches so that you don't have to worry about the order things are included.

UNITY_COUNTER_TYPE

The internal counters which count the number of failures, tests, and ignores are by default unsigned shorts... change it to something else as appropriate... just don't blame us when you write test 256 and your unsigned char seems to give you weird results.

UNITY_INCLUDE_64

Define this to include 64-bit support... otherwise only 8-32 bit words will be supported. There is a significant size and speed impact to enabling 64-bit support, so don't define it if you don't need it.

UNITY_INT_WIDTH
Define this to something other than the default 32 if you're working on a system with larger or smaller ints.

UNITY_EXCLUDE_FLOAT

Don't include the floating point support... useful for those smaller micros where you don't want to include floating point libraries.

UNITY_FLOAT_PRECISION

This how close the floats need to be in order to be considered “equal”. It defaults to 0.00001f

UNITY_FLOAT_TYPE

This defaults to float... but maybe you want double? Double double?

UNITY_LINE_TYPE

This defaults to an unsigned short... but if you have huge files (greater than 65535) you may need to raise it. You could save a bit of memory if your files are all less than 255 and you set this to char.

UNITY_LONG_WIDTH

This is here to primarily figure out what kind of 64-bit support you have... in the end it's using this to figure out if you need to specify a long (set to 64) or a long long (set to 32).

UNITY_POINTER_WIDTH

By default we're assuming your pointers are 32 bits wide... but if you're running something swank it might be 64, or if you're running something tiny it might be 16. If you're getting ugly compiler warnings about casting from pointers, this is the one to look at.