(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, 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

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 exe 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. |
| `TEST_WRAP(function)` | If the test functions call helper functions and those helper functions have the ability to make assertions, calls to those helpers should be wrapped in a TEST_WRAP macro. This macro aborts the test if the helper triggered a failure. |

## *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_THROW can then be used at any time within the tests to return to the last TEST_PROTECT call. This will require a longjmp library to exist for your platform.

| | |
|---|---|
| TEST_PROTECT() | Setup and Catch macro |
| TEST_THROW (message) | Abort Test macro |

*Example*:

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

If MyTest calls TEST_THROW, a failure with the message provided will be inserted, and 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*

| | |
|---|---|
| `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. |
| `TEST_ASSERT_EQUAL_UINT (expected, actual)` | Compare two integers for equality and display errors as unsigned integers. |
| `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_HEX (expected, actual)` | Another way of calling `TEST_ASSERT_EQUAL_HEX32` |
| `TEST_ASSERT_INT_WITHIN (delta, expected, actual)` | Asserts that the actual value is within plus or minus delta of the expected value. |
| `TEST_ASSERT_EQUAL_MESSAGE (expected, actual, message)` | Another way of calling `TEST_ASSERT_EQUAL_INT_MESSAGE` |
| `TEST_ASSERT_EQUAL_INT_MESSAGE (expected, actual, message)` | Compare two integers for equality and display errors as signed integers. Outputs a custom message on failure. |
| `TEST_ASSERT_EQUAL_UINT_MESSAGE (expected, actual, message)` | Compare two integers for equality and display errors as unsigned integers. Outputs a custom message on failure. |
| `TEST_ASSERT_EQUAL_HEX8_MESSAGE (expected, actual, message)` | Compare two integers for equality and display errors as an 8-bit hex value. Outputs a custom message on failure. |
| `TEST_ASSERT_EQUAL_HEX16_MESSAGE (expected, actual, message)` | Compare two integers for equality and display errors as an 16-bit hex value. Outputs a custom message on failure. |
| `TEST_ASSERT_EQUAL_HEX32_MESSAGE (expected, actual, message)` | Compare two integers for equality and display errors as an 32-bit hex value. Outputs a custom message on failure. |
| `TEST_ASSERT_EQUAL_HEX_MESSAGE (expected, actual, message)` | Another way of calling `TEST_ASSERT_EQUAL_HEX32_MESSAGE` |

### *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. |

### *String Assertions*

| | |
|---|---|
| `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 |

# 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.