

UNITY: A LIGHTWEIGHT C TEST HARNESS FOR EMBEDDED SYSTEMS

ABSTRACT

One of the key benefits on the Agile movement is moving the test activity from test-after-construction (TAC) to test-before-construction (TBC). However almost all current test frameworks are based around either Java or C++. This paper introduces Unity, an open source lightweight test harness that can be used for in-target testing of an embedded C application. This paper explains how Unity works, how to integrate it into your environment and how to use it.

Niall Cooling

Feabhas Limited

5 Lowesden Works

Lambourn Woodlands

Hungerford, RG17 7RY, UK

skype: feabhas

twitter: @feabhas

blog.feabhas.com

linkedin: niallcooling

1 INTRODUCTION

1.1 AGILE MOVEMENT

1.1.1 TEST DRIVEN DESIGN

The last decade of software development has seen the growth of *Agile* software development. Acceptance of Agile software development has widened within application development and is now becoming accepted in Embedded Development. However there still can be confusion as to what 'Agile' actually entails.

There are ongoing debates about the merits and shortcomings of Agile processes, but one of the major benefits has been around the concept of Test-Driven Design (TDD) (1). TDD advocates writing test code before application code, thus bringing all the concepts of structured testing to the average programmer (rather than being seen as a separate activity)¹; an alternative and less political term for this is "Test-Before-Construction" (TBC).

1.2 TOOLS

1.2.1 XUNIT

Programmers being programmers have developed tools to automate and simplify the process of TBC. A major part of the TBC process is the use of a test harness/framework for the running and reporting of tests.

Testing tools have existed for many years but have typically been based on the more traditional model of "Test-After-Construction" (TAC). What makes the TBC approach different is that it is a grammatical model developed by the programmer (i.e. they actually write the software in the same programming language they're testing in) as opposed to a separate toolchain and possibly a separate programming paradigm (e.g. TSL (2)).

There are now many code-driven test frameworks for all manner of languages, which collectively fall under the umbrella term of "XUnit" (3). Their origins begin with Smalltalk but their popularity grew with the development of the JUnit (4) framework for testing Java. Most modern languages (Python, Ruby, etc.) now have their own test framework closely based around the ideas and concepts from JUnit.

With the popularity of Agile development and XUnit test frameworks, programmers have looked to use this approach with older languages, most notably C++. There now exist over 20 different C++ testing frameworks based around the XUnit model, including CppUnit (5), Google's own C++ Testing Framework (6) and CppUTest (7) (specifically targeted at embedded systems testing).

1.2.2 ISSUES

¹ Again, there are debates around the reality of TDD in non-trivial, non-software-only systems, but we can save that for a later date

However, as C is still the dominant programming language for embedded systems (8) our interest is in using a framework to test embedded C applications. The majority of leading frameworks prove problematic when looking at testing in the context of an embedded application. Of course it is possible to test a C program using a C++ test framework, but this isn't always suitable due to the change of programming paradigm or due to a lack of compiler support.

Another problem for the embedded C programmer is that many, if not most, frameworks have been developed by the open source community. This is not a criticism of open source, but it means these tools are heavily dependent on the Linux operating system and the GCC toolchain. A knock-on issue is that many frameworks are library based, which means running them in a non-GCC environment (especially with a cross-compiler) can be a frustrating, and sometimes futile, porting activity.

1.2.3 GOALS

So rather than re-inventing the wheel we were looking for a test framework with some clear goals:

- C - not C++
- Simple - ideally not library based
- Embeddable – does not require stdout, but supports redirection of output over serial port
- Small footprint – systems with small amounts (kB) of SRAM
- Not compiler-specific - works with commercial cross compilers
- Works on any OS – especially Windows
- Single toolchain – do not have to work outside the regular IDE

2 UNITY

2.1 OPTIONS FOR C

Looking around, there are a number of commercial and open source options for C-based test frameworks, with CUnit (9) being the best known of these. However, CUnit is a classic "Linux-centric" toolset that requires the building of a library. There are ports for Visual Studio, but no obvious support for embedded compilers.

I cannot say I have extensively looked at all the C-based test frameworks (this is not meant to be a comparison of test frameworks), but upon discovering Unity (10) it appeared to tick all the boxes, so we started to use it in anger.

2.2 UNITY

2.2.1 ATTRACTIONS

First and foremost, the immediate attraction of Unity is its simplicity. Embedded programmers are busy people with constant deadlines, so I believe a crucial point for acceptance of new ideas or concepts is how easily it fits with the way people develop code today. Being code-based testing it means C programmers are writing their tests in C; a point that shouldn't be underestimated. Admittedly, an OO/C++ framework can offer a richer feature set, but a major goal is for the programmer to embrace testing. Writing the tests in the same language and tool environment that the development is developed in eliminates the barrier of having to 'context switch' to another tool and way of thinking in order to test their code.

Secondly, the code of Unity is in simple source code form, consisting of two header files and one C file. To use Unity involves including one of the headers and setting up the compilation/build paths to link to the Unity files

(more on this later). The code base is also compiler independent, meaning no porting activity is required to work with any standard C compiler. Note, however, much of the automation around Unity is based on Ruby scripts. Ruby isn't required, but as you shall see makes Unity a really powerful test framework.

Finally, and probably its greatest attraction, is that Unity was developed by embedded C programmers (11).

2.2.2 TESTING BASICS

Before getting in to the details of Unity, it should be noted that the XUnit frameworks and their derivatives are all focused on unit testing (12). The focus of the testing is on the functionality of the application rather than performance, stress, etc. This means invoking C function(s) via their interface (declaration) and looking for conditions to be true or false up on executing the function(s).

The XUnit frameworks introduce (or formalize) certain concepts.

First is the four phases of a test:

- Setup
- Test – run the test
- Analyze – report on the test outcome
- Teardown

The Setup and Teardown phases are responsible for putting the test environment in to a well-known state prior to running one or more tests so that test results are repeatable – known as the *test context*, or more commonly in the Agile world, the *test fixture*. Depending on the framework, tests may share the same fixture, or each test may have its own fixture. In Unity all tests share the same fixture setup and teardown.

Finally we need a *Test runner*, a part of the framework that executes the tests.

2.2.3 REQUIREMENTS

To explain setting up and using Unity 2.0 the examples are based on using IAR EW (13) under Windows 7.

Download the Unity zip file from Sourceforge and unzip it to a known location. I recommend, and will base my examples on `C:\unity`, but the files can reside anywhere. I have found that when setting up path names and upgrading unity the simple path makes life easier.

Once unzipped, our initial files of interest can be found under `C:\unity\src`. Here we will find:

- `unity.c`
- `unity.h`
- `unity_internals.h`

You will also find `testHarness.c`, but ignore that for now. We are now ready to try out Unity.

SIMULATED C PROJECT AND UNITY

Create a new C project (e.g. unity1) and import the Unity C file into the project.

[IAR] To add the unity.c source file to the project, it is probably best to create a new group for the Unity files. Select Project->New Group... and name your group (e.g. unity) then select Project->Add Files.. and select unity.c from the unity\src directory (e.g. C:\unity\src).

First we need to include "unity.h" in our "main.c" file.

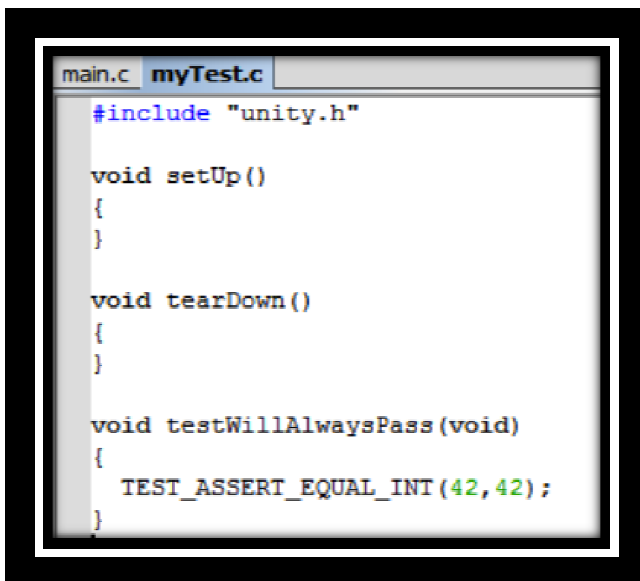
If we build the project at this point the compilation will fail as the project does not know the path for the unity header file. This involves setting the C preprocessor include path so it can see the unity header files.

[IAR] To add the correct path, open the project's options (ALT+F7), select "C/C++ Compiler" and select the tab "Preprocessor". Add the unity path to the "Additional include directories:" text box (e.g. C:\unity\src)

Rebuild the project and this should now build without errors if the path is set up correctly.

Next we need to build a simple test. Create a new C file (e.g. myTests.c) and add this to the project. Enter the following simple test code to the test file. This is using the simple Unity test to compare two integers. All Unity test functions must be of the form:

```
void test<the test name>(void)
```



```
main.c myTest.c
#include "unity.h"

void setUp()
{
}

void tearDown()
{
}

void testWillAlwaysPass(void)
{
    TEST_ASSERT_EQUAL_INT(42, 42);
}
```

Note we need to create (empty) setup and teardown functions as the unity framework expects these functions (without them it will fail to link). Save the code and return to main.c.

We now need to build the simple test runner part of the Unity test harness in main.c. Unity uses setjmp/longjmp to manage the tests, so we need to include setjmp.h from the C standard library. By default the reporting mechanism of Unity uses the function putchar from stdio.h (this can be redirected). Note that this requirement for setjmp/longjmp means that on small 8-bit processors with a hardware stack, Unity is unlikely to work (check your compiler documentation).

We declare a function prototype for our test function from our test file (i.e. testWillAlwaysPass from myTest.c). We could define a header for myTest.c but here it is just as simple to using an extern declaration (I know the extern isn't needed but it is good practice if the prototype is from a function in another file to include it).

Unity requires you to define a function called runTest that takes a function pointer as a parameter. The following code is a minimum definition for a Unity runTest function (we will see the full code later). The TEST_PROTECT macro wraps the use of setjmp/longjmp to manage tests that fail or are aborted.

```
main.c myTest.c
#include "unity.h"
#include <stdio.h>
#include <setjmp.h>

extern void testWillAlwaysPass(void);

void runTest(UnityTestFunction test)
{
    if(TEST_PROTECT())
    {
        test();
    }
}

int main()
{
    UnityBegin();
    RUN_TEST(testWillAlwaysPass, __LINE__);
    UnityEnd();
    return 0;
}
```

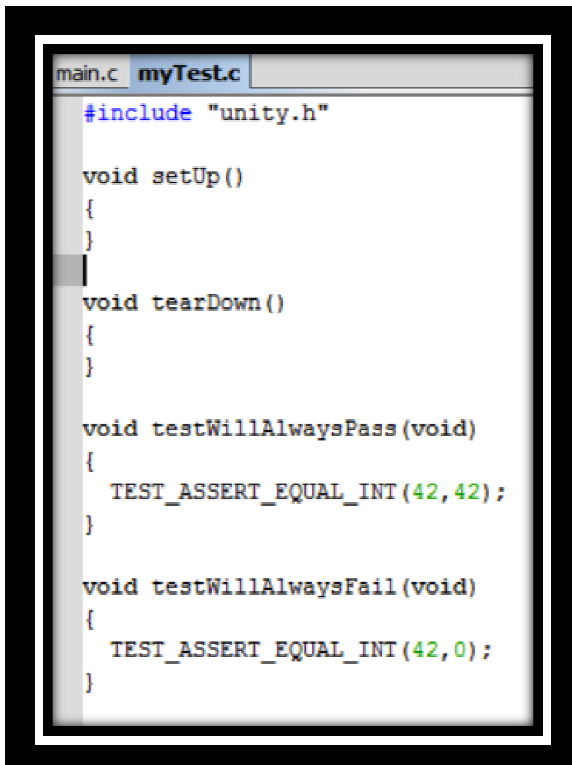
We are now in a position to run the Unity test. The output from the Unity test will now be displayed in the Terminal I/O window.

```
Terminal I/O
Output:
:18:testWillAlwaysPass:PASS
-----
1 Tests 0 Failures 0 Ignored
OK
```

ADDING ANOTHER TEST

To define another test the following steps are required.

First, in the test file create a new function with a new test name. The test we've shown here will fail.



```
main.c myTest.c
#include "unity.h"

void setUp()
{
}

void tearDown()
{
}

void testWillAlwaysPass(void)
{
    TEST_ASSERT_EQUAL_INT(42, 42);
}

void testWillAlwaysFail(void)
{
    TEST_ASSERT_EQUAL_INT(42, 0);
}
```

Update main.c as follows

```
main.c | myTest.c
#include "unity.h"
#include <stdio.h>
#include <setjmp.h>

extern void testWillAlwaysPass(void);
extern void testWillAlwaysFail(void);

void runTest(UnityTestFunction test)
{
    if(TEST_PROTECT())
    {
        test();
    }
}

int main()
{
    UnityBegin();
    RUN_TEST(testWillAlwaysPass, __LINE__);
    RUN_TEST(testWillAlwaysFail, __LINE__);
    UnityEnd();
    return 0;
}
```

```
Terminal I/O
Output: Log file:
:19:testWillAlwaysPass:PASS
:18:testWillAlwaysFail:FAIL: Expected 42 Was 0
-----
2 Tests 1 Failures 0 Ignored
FAIL
```

As you can see, adding new tests is a repetitive process:

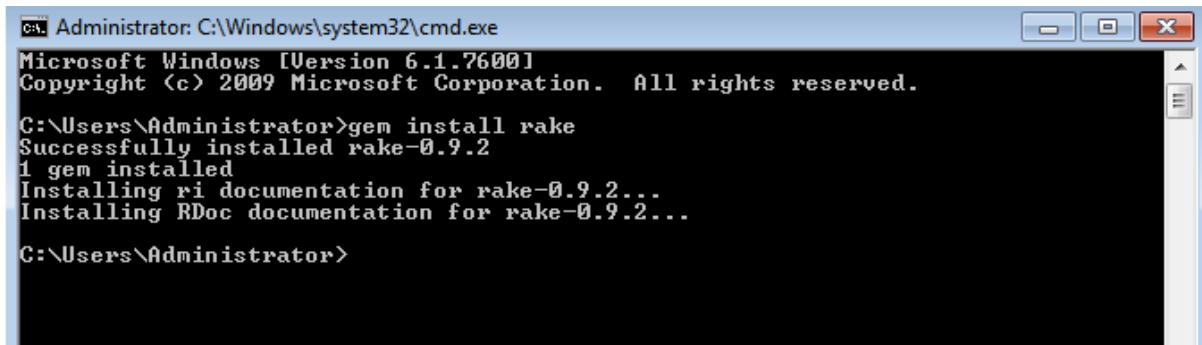
- Write the new test of the form `void testName(void)`
- Declare the test function in `main.c`
- Add `RUN_TEST` to the main function to run the test function.

Anywhere we have repetition we should look to automate. Unity supplies Ruby scripts to auto-generate the required `main.c` file.

INSTALLING RUBY

Installing Ruby for Windows is very straightforward thanks to the Ruby Installer executable that sets up the `PATH` environment as well.

Once installed, open a command window and enter `gem install rake` at command line. Rake is Ruby's make and a requirement of Unity.



```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>gem install rake
Successfully installed rake-0.9.2
1 gem installed
Installing ri documentation for rake-0.9.2...
Installing RDoc documentation for rake-0.9.2...

C:\Users\Administrator>
```

GENERATING TEST RUNNER

Rather than manually editing the main (test runner) file each time we add a new test, Unity uses a Ruby script to parse the test code and automatically generate the test runner main file. To automatically generate a test runner.c file, open a command window and navigate to the project containing the unity test file (e.g. `myTest.c` from previous example).

At the command line type:

```
Ruby c:\unity\auto\generate_test_runner.rb myTest.c
```

If this runs successfully it will create a new file of the form: `<test_file_name>_Runner.c` for example, from `myTest.c` we get `myTest_Runner.c`

We now need to include this in our project. If we build now we will get a link time error (duplicate definitions for "main") as both `main.c` and `myTest_Runner.c` both define a main function. We need to remove/exclude `main.c` from our build. The project should now rebuild without and errors; run the code as before.

STRUCTURE OF THE AUTO-GENERATED RUNNER FILE

The auto-generated Runner file should not be edited (as it will be overwritten when new tests are generated and this file needs regenerating). If you look at the file there shouldn't be anything of surprise in there. The `RUN_TEST` is configured as a macro, and there are some additional items set up for reporting purposes; but all in all it is very similar to the `main.c` we developed earlier.

One difference is the second parameter to the `RUN_TEST` macro. In our previous code we used `__LINE__` just as filler. However, the actual number supplied is the line number of the test function in `myTest.c` (e.g. `testWillAlwaysPass` is at line 11 in `myTest.c`).

```
//=====MAIN=====
int main(void)
{
    Unity.TestFile = "myTest.c";
    UnityBegin();
    RUN_TEST(testWillAlwaysPass, 11);
    RUN_TEST(testWillAlwaysFail, 16);

    return (UnityEnd());
}
```

However the real benefit comes when we add a new test.

REGENERATING THE TEST RUNNER

If we now want to add a new test it involves the following steps:

1. Edit myTest.c and create test function
2. Regenerate the test runner myTest_Runner.c using the Ruby script
3. Rebuild and execute

If we add the following test to myTest.c

```
void testBinaryMatchPass(void)
{
    TEST_ASSERT_BIT_HIGH(4, 0x10);
}
```

Run the Ruby script:

Ruby c:\unity\auto\generate_test_runner.rb myTest.c

Rebuild and run the output is:

```
Terminal I/O
Output: Log file
myTest.c:11:testWillAlwaysPass:PASS
myTest.c:18:testWillAlwaysFail:FAIL: Expected 42 Was 0
myTest.c:21:testBinaryMatchPass:PASS
-----
3 Tests 1 Failures 0 Ignored
FAIL
```

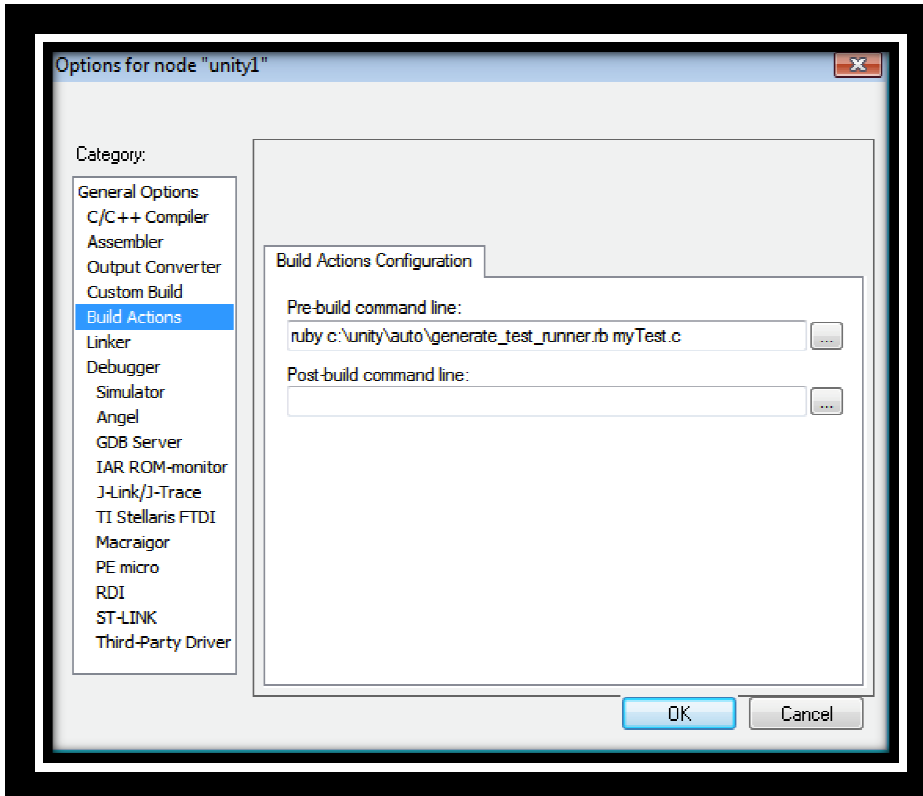
Revisiting the test runner file, myTest_Runner.c the automated changes are the inclusion of the extern function prototype testBinaryMatchPass

```
//=====External Functions This Runner Calls=====  
extern void setUp(void);  
extern void tearDown(void);  
extern void testWillAlwaysPass(void);  
extern void testWillAlwaysFail(void);  
extern void testBinaryMatchPass(void);
```

and the update of the main function to call RUN_TEST(testBinaryMatchPass, 21).

```
//=====MAIN=====  
int main(void)  
{  
    Unity.TestFile = "myTest.c";  
    UnityBegin();  
    RUN_TEST(testWillAlwaysPass, 11);  
    RUN_TEST(testWillAlwaysFail, 16);  
    RUN_TEST(testBinaryMatchPass, 21);  
  
    return (UnityEnd());  
}
```

Depending on your environment, one further improvement can be made. Remembering to switch to the command line and auto-generate the revised test runner file each time can be tedious and error-prone. Many environments have a "pre-build" command line option. For example, in IAR EW there is the option for pre-build actions. Here we can run the Ruby script (though please note I have had some problems regarding confliction between IAR and Windows UAC which causes the pre-build to fail; I have yet to get a satisfactory workaround).



2.2.4 RUNNING ON TARGET

Overall, un-optimised Unity adds about 1kB of code and about 90 bytes of data plus the standard library requirements. In terms of running Unity in a target system, it is pretty straightforward.

Many modern debug environments using JTAG connections support automatic redirection of standard I/O back to the debug IDE (often referred to as semi-hosting).

In the target environment either semi-hosting may not be available or we want to eliminate, for size reasons, `stdio.h`. Unity allows redirection of the test summary messages (typically to a serial port) by implementing a function and mapping the macro `UNITY_OUTPUT_CHAR` to it. This is managed in `unity_internals.h` (which is included in `unity.h`) so define the macro before including the `unity.h` header.

```

//-----
// Output Method
//-----

#ifndef UNITY_OUTPUT_CHAR
//Default to using putchar, which is defined in stdio.h above
#define UNITY_OUTPUT_CHAR(a) putchar(a)
#else
//If defined as something else, make sure we declare it here so it's ready for use
extern int UNITY_OUTPUT_CHAR(int);
#endif
    
```

TESTING WITH UNITY

Unity supplies a wide range of macros for basic unit testing. Most are quite straightforward, but still eliminate reinventing the wheel. The list can be found in `unity.h`, and include examples such as

- Simple Boolean - `TEST_ASSERT_TRUE`, `TEST_ASSERT_FALSE`
- Standard integer - `TEST_ASSERT_EQUAL`, `TEST_ASSERT_NOT_EQUAL`
- Size specific – `TEST_ASSERT_EQUAL_INT8`, `_INT16`, `_INT32`
- Sign specific – `TEST_ASSERT_EQUAL_UINT8`, `_UINT16`, ...
- Base specific – `TEST_ASSERT_EQUAL_HEX8`, `_HEX16`, ...
- Bit masks - `TEST_ASSERT_BITS`, `TEST_ASSERT_BITS_HIGH`, `TEST_ASSERT_BIT_LOW`
- Ranges – `TEST_ASSERT_INT_WITHIN`
- Arrays – `TEST_ASSERT_EQUAL_INT_ARRAY`
- Strings and structures – `TEST_ASSERT_EQUAL_PTR`, `_STRING`, `_MEMORY`
- Floating point (if enabled) – `TEST_ASSERT_FLOAT_WITHIN`

All tests can include an extra user definable message

```
void testWillAlwaysFail(void)
{
    TEST_ASSERT_EQUAL_INT_MESSAGE(42, 0, "Duh!");
}
```

Which is included in the output if the test fails.

```
terminal I/O
Output:
myTest.c:11:testWillAlwaysPass:PASS
myTest.c:18:testWillAlwaysFail:FAIL: Expected 42 Was 0. Duh!
myTest.c:21:testBinaryMatchPass:PASS
-----
3 Tests 1 Failures 0 Ignored
FAIL
```

An example for a simple Unity target test is shown, where a seven segment display is being driven off GPIO lines 16 through 19 on Port 1 of an NXP LPC2129 ARM7 system. The test code checks that the appropriate bits in the Direction register are set high after the device has been initialised.

```

sevenSegTest.c
#include "unity.h"
#include "nxp/iolpc2129.h"

#include "seven_seg.h"

void setUp()
{
    IO1DIR &= ~0x000F0000;
}

void tearDown()
{
}

void sevenSegInitTest(void)
{
    seven_seg_init();
    TEST_ASSERT_BITS_HIGH(0x000F0000, IO1DIR);
}

```

3 BEYOND UNITY

3.1 ISOLATING THE UNIT UNDER TEST

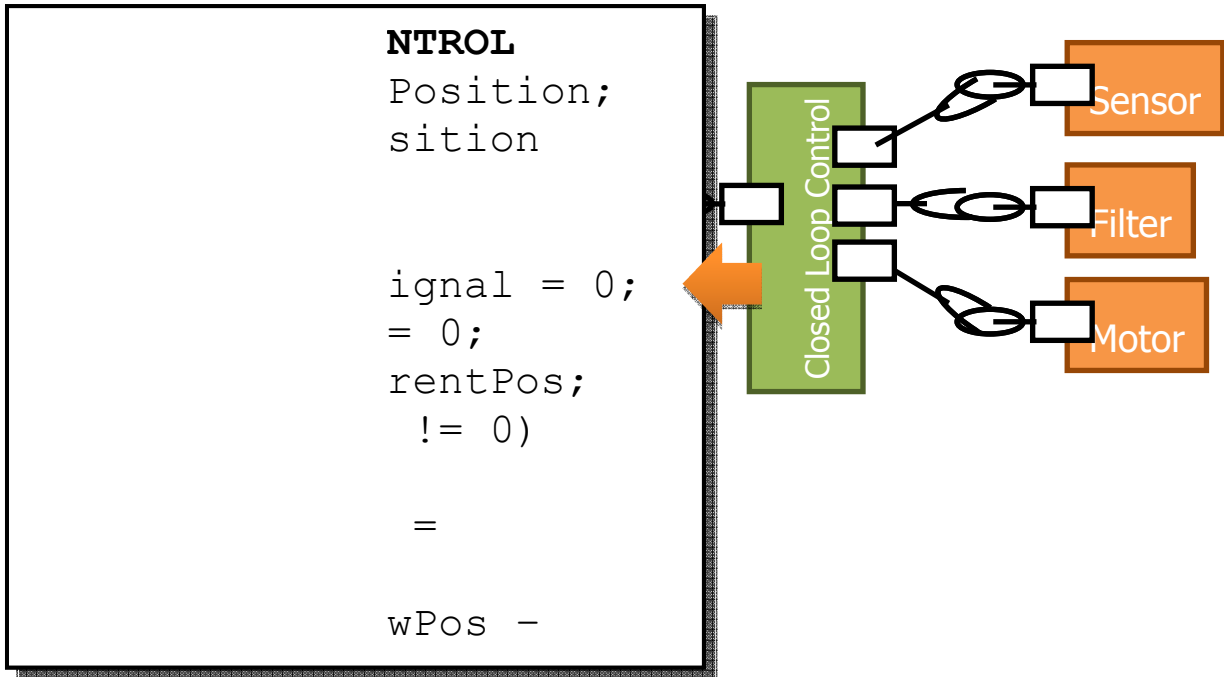
Until now we considered modules in isolation; and the units under test only have a single interface which the test harness can exploit. In practice, modules typically communicate with other modules. The public interface of a module is therefore the combination of:

- The Provided Interface. This is the services the module provides to its clients
- The Required Interface. This is the set of calls the module will make to its peers or subordinates in order to fulfil its function.

One of the problems with unit testing is that it can be difficult to isolate the piece of code required for testing (referred to as the *Unit Under Test* or the UUT).

For testing purposes subordinate or peer code can be replaced by *stub* code. The stub has the same interface as the actual system code but provides a *simulation* of its behaviour. Typically, a stub produces a fixed response or a narrow range of responses; as long as the responses are adequate to test the behaviour of the unit under test. The use of stubs to support higher-level code leads to them sometimes being referred to as *scaffold code*. Stubs can be replaced by working code as development continues.

What happens if the unit-under-test is a control object (doesn't return values)? In this example the Closed Loop Control element provides a closed-loop control mechanism (for example, for a motor position controller). The Closed Loop Control unit requires the services of a Motor unit (for drive) a Sensor unit (for feedback) and a Filter unit (for calculating drive signals). The four units form a hierarchical composite.



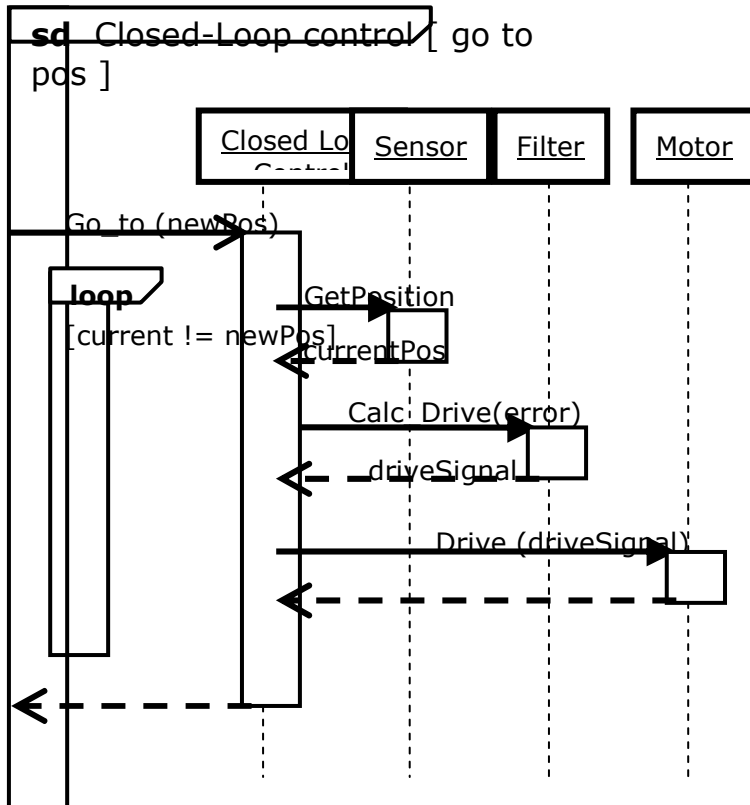
However, the interface of Closed Loop Control has only one function – Go_to(). This takes a single parameter (position) and returns no values.

```

void test_Go_To_1_0_Test(void)
{
    // How do I test this???
    Go_To(1.0);
}
    
```

We could test Closed Loop Control with stubs for the Sensor, Motor and Filter, but what tests could we run? Closed Loop Control is an example of where data-oriented testing can be of limited value.

Given this sequence diagram showing control behaviour, the behaviour of the Closed Loop Control object can only really be verified by examining how it affects its subordinates.



That is, the Closed Loop Control object is functioning correctly *if* it makes particular function calls, in a particular order, and with particular values. If any of these is incorrect, then the behaviour of the Closed Loop Control object must be incorrect. This form of testing is referred to as **Behavioural Testing**.

In this case we cannot substitute traditional stub objects for the Sensor, Motor and Filter objects.

3.2 STUBS WITH EXPECTATIONS

For a control-like object we must verify its behaviour from the point of view of its subordinates. That is, did the unit under test make the correct calls to its subordinates, with the right data, at the right time?

Mock objects are an extension of traditional function stubs. A *Mock* object 'stands in' for a real module. The *Mock* object has the same interface as the actual system code; and can return representative values. However, its interface is extended with additional functions for testing purposes.

When performing behavioural testing using *Mock* objects, it is the *Mock objects* that confirm the behaviour of the unit under test, *not* the test context code. The Test Context is responsible for configuring the *Mock* objects and invoking the behaviour of the unit under test. The *Mock* objects tell the Test Context whether the unit under test is working as expected; the Test Context merely collates the results.

A *mock* object can be set *expectations*. An expectation is a function call the *mock* object is expected to receive. Expectations are set in sequential order. Expected calls can also be given expected parameters; and can even be given a particular response to return to the caller. If a *mock* object has all its expectations fulfilled it passes the test; if not all expectations have been fulfilled the unit under test must have failed.

3.3 CMOCK

Unity has mocking support through CMock (14) - a collection of files, including Ruby scripts to generate the mocks. CMock is a nice little extension to Unity which takes your header files and creates a Mock interface for it so that you can easily unit test modules that rely on other modules.

First download CMock zip file from SourceForge. Unzip this to a well-known location, e.g. `c:\cmock`. In the docs folder you will find a brief introduction to CMock and configuration details. The `lib` directory contains all of the interesting Ruby scripts and the `src` directory contains `cmock.c` and `cmock.h`.

As with Unity, the `cmock.c` needs adding to the project and the project path needs to be setup to see `cmock.h`.

To test the `Go_To` function, we need to create mocks for `get_position`, `calc_drive` and `drive` functions. These are contained in the headers `sensor.h`, `filter.h` and `motor.h` respectively.

CREATING MOCKS

Before creating the mock objects, create a folder to store them in (keeping them separate from the main project files). By default the Ruby scripts use a subdirectory called "mocks".

Use the supplied Ruby scripts to create the mock files for the required headers by typing: `cmock.rb <header files>`

e.g.

```
c:\cmock\lib\cmock.rb sensor.h filter.h motor.h
```



```
C:\Users\Administrator\Documents\programming\ESC Boston>ruby c:\cmock\lib\cmock.
rb sensor.h filter.h motor.h
Creating mock for sensor...
Creating mock for filter...
Creating mock for motor...

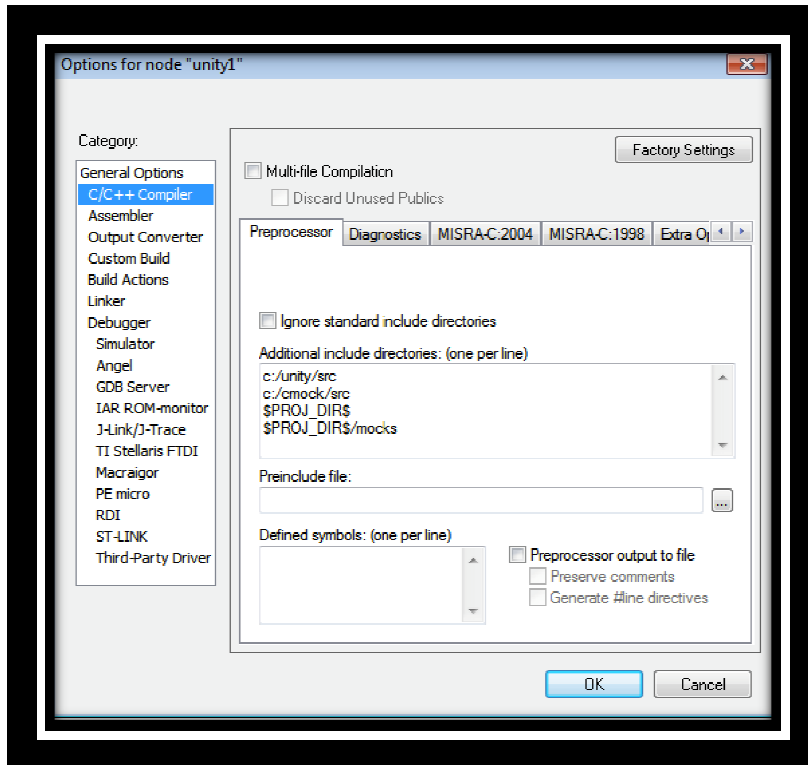
C:\Users\Administrator\Documents\programming\ESC Boston>
```

In the mocks subdirectory you will find a `Mock<header>.c` and `Mock<header>.h` file for each supplied header



USING MOCKS

Create a group in the project and add the (generated) mock .c files to the project. Modify the preprocessor settings so the build knows where the mocks are (\$PROJ_DIR\$\mocks) and also add the project directory (\$PROJ_DIR\$) so the mock files can find the original headers (e.g. sensor.h)



TESTING WITH MOCKS

To use the mocks we have to include the key header files required for the test in our test harness file (controller_test_harness.c) e.g.

```
#include "Mocksensor.h"
#include "Mockfilter.h"
#include "Mockmotor.h"
```

For each of our stub functions, the Ruby scripts have created mock functions based on the signature of the original function. We can then test the 'Go-To' function thus:

```
void test_Go_To_1_0_Test(void)
{
    // Now testing with mocks
    Go_To(1.0);
}
```

If we build and run our test with the mocks, the output is:

```

Terminal I/O
Output:
controller_test_harness.c:14:testControllerTest:FAIL: Function 'get_position' called more times than expected.
-----
1 Tests 1 Failures 0 Ignored
FAIL
|
    
```

The test reported that the function “get_position”, declared in sensor.h has been called (as expected), but for this test to pass we have to ‘program’ the mock framework to say that we expect this function (and calc_drive() and drive()) to be called during this test.

For each function declared in sensor.h a mock (stub) function is created (e.g. the stub of *float get_position(void)* is automatically created). In addition, the mock framework creates another function that allows us to inform it that we expect this function to be called by the test. The mock-expectation function created, takes the form

```
void get_position_ExpectAndReturn(float cmock_retval)
```

The ‘cmock_retval’ is an argument we supply, that the stub of get_position will actually return. So here is an example of testing ‘Go_To’ using the CMock framework:

```

void test_Go_To_1_0_Test(void)
{
    get_position_ExpectAndReturn(1.0); // will give error of 0.0
    calc_drive_ExpectAndReturn(0.0, 1.0); // error = 0.0, drive_signal = 1.0
    drive_Expect(1.0); // drive_signal = 1.0
    Go_To(1.0);
}
    
```

Given the earlier code, passing the Go_To function and argument of 1.0, and the said expectation, the test will now pass.

```

Terminal I/O
Output:
controller_test_harness.c:18:test_Go_To_1_0_Test:PASS
-----
1 Tests 0 Failures 0 Ignored
OK
    
```

Note that if we state we expect a function to be called and it is not, or it is called more times than expected, both these cases will result in test failure. For example, if we wanted to iterate around the loop a couple of times checking the algorithmic behaviour, we could write our test code thus:

```
void test_Go_To_2_0_Test(void)
{
    get_position_ExpectAndReturn(1.0);    // will give error of 1.0
    calc_drive_ExpectAndReturn(1.0, 1.5); // error = 1.0, drive_signal = 1.5
    drive_Expect(1.5);                   // drive_signal = 1.5
    get_position_ExpectAndReturn(2.0);    // will give error of 0.0
    calc_drive_ExpectAndReturn(0.0, 1.0); // error = 0.0, drive_signal = 1.0
    drive_Expect(1.0);                   // drive_signal = 1.0
    Go_To(2.0);
}
```

As you can imagine, a test could be set up using real test vectors stored in a file (if available).

```
Terminal I/O
Output:
controller_test_harness.c:18:test_Go_To_1_0_Test:PASS
controller_test_harness.c:26:test_Go_To_2_0_Test:PASS
-----
2 Tests 0 Failures 0 Ignored
OK
```

Mocks can do a lot more than covered here (e.g. calling user-defined code as part of the stub call – a concept known as a *wrapper*). One word of warning: mock frameworks (including CMock) are dependent on dynamically-allocated memory, so may not be useable in smaller embedded systems. That said, mocks are more useful in a host environment to simulate (and stub) the behaviour of low-level software, or even hardware devices themselves. In such cases, dynamic memory allocation may not be an issue.

4 SUMMARY

Looking back at the original goals, Unity fits these perfectly:

- C - not C++
- Simple - ideally not library based
- Embeddable – does not require stdout, but supports redirection of output over serial port
- Small footprint – systems with small amounts (kB) of SRAM
- Not compiler-specific - works with commercial cross compilers
- Works on any OS – especially Windows
- Single toolchain – do not have to work outside the regular IDE

With the addition of CMock (and not to mention the other projects of CException and Ceedling) we have a very useful and usable testing framework for the embedded C programmer. I would encourage you to download these tools and give them a try. For more information about TDD and Embedded C then I would recommend James Grenning's Book 'Test-Driven Development for Embedded C' (15).

BIBLIOGRAPHY

1. **Beck, Kent.** *Test Driven Development: By Example*. s.l. : Addison-Wesley Professional, 2002. 0321146530.
2. HP WinRunner. *Wikipedia*. [Online] http://en.wikipedia.org/wiki/HP_WinRunner.
3. Xunit. *Martin Fowler*. [Online] <http://www.martinfowler.com/bliki/Xunit.html>.
4. Resources for Test driven Development. *JUnit.org*. [Online] <http://www.junit.org/>.
5. CppUnit Wiki. *sourceforge*. [Online] http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page.
6. googletest. [Online] Google. <http://code.google.com/p/googletest/>.
7. CppUTest Core Manual. *CppUTest* . [Online] <http://www.cpputest.org/>.
8. What languages do you use to develop software? . *VDC*. [Online] http://blog.vdcresearch.com/embedded_sw/2010/09/what-languages-do-you-use-to-develop-software.html.
9. CUnit: A Unit Testing Framework for C. *sourceforge*. [Online] <http://cunit.sourceforge.net/>.
10. Unity Intro. *throw the switch*. [Online] <http://throwtheswitch.org/white-papers/unity-intro.html>.
11. Atomic Embedded. *Atomic Object*. [Online] <http://www.atomicobject.com>.
12. Unit Testing. *Wikipedia*. [Online] http://en.wikipedia.org/wiki/Unit_testing.
13. IAR Embedded Workbench® for ARM. *IAR Systems*. [Online] <http://www.iar.com/website1/1.0.1.0/68/1/>.
14. Cmock Intro. *throw the switch*. [Online] <http://throwtheswitch.org/white-papers/cmock-intro.html>.
15. Test Driven development for Embedded C. *The Pragmatic Bookshelf*. [Online] 2011. <http://pragprog.com/book/jgade/test-driven-development-for-embedded-c>. ISBN: 978-1-93435-662-3.



TRAINING IN REAL-TIME
EMBEDDED DEVELOPMENT

Feabhas Ltd

5 Lowesden Works
Lambourn Woodlands, Hungerford
Berkshire RG17 7RY, UK

Tel: +44(0)1488 73050

Fax: +44(0)1488 73051

Email: info@feabhas.com

Web: www.feabhas.com