Chapter 60: Testing frameworks

Many developers use unit tests to check that their software works as expected. Unit tests check small units of larger pieces of software, and ensure that the outputs match expectations. Testing frameworks make unit testing easier by providing set-up/tear-down services and coordinating the tests.

There are many unit testing frameworks available for C. For example, Unity is a pure C framework. People quite often use C++ testing frameworks to test C code; there are many C++ test frameworks too.

Section 60.1: Unity Test Framework

Unity is an xUnit-style test framework for unit testing C. It is written completely in C and is portable, quick, simple, expressive and extensible. It is designed to especially be also useful for unit testing for embedded systems.

A simple test case that checks the return value of a function, might look as follows

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

A full test file might look like:

```
#include "unity.h"
#include "UnitUnderTest.h" /* The unit to be tested. */

void setUp (void) {} /* Is run before every test, put unit init calls here. */
void tearDown (void) {} /* Is run after every test, put unit clean-up calls here. */

void test_TheFirst(void)
{
    TEST_IGNORE_MESSAGE("Hello world!"); /* Ignore this test but print a message. */
}

int main (void)
{
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst); /* Run the test. */
    return UNITY_END();
}
```

Unity comes with some example projects, makefiles and some Ruby rake scripts that help make creating longer test files a bit easier.

Section 60.2: CMocka

CMocka is an elegant unit testing framework for C with support for mock objects. It only requires the standard C library, works on a range of computing platforms (including embedded) and with different compilers. It has a tutorial on testing with mocks, API documentation, and a variety of examples.

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}
```

```
void null_test_fail (void ** state)
{
    assert_true (0);
}

/* These functions will be used to initialize
   and clean resources up after each test run */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
}


int main (void)
{
    const struct CMUnitTest tests [] =
    {
        cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
    };

    /* If setup and teardown functions are not
       needed, then NULL may be passed instead */

    int count_fail_tests =
        cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}
```

Section 60.3: CppUTest

CppUTest is an xUnit-style framework for unit testing C and C++. It is written in C++ and aims for portability and simplicity in design. It has support for memory leak detection, building mocks, and running its tests along with the Google Test. Comes with helper scripts and sample projects for Visual Studio and Eclipse CDT.

```
#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>


TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}

/* Test runner may be provided options, such
   as to enable colored output, to run only a
   specific test or a group of tests, etc. This
   will return the number of failed tests. */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}
```

A test group may have a `setup()` and a `teardown()` method. The `setup` method is called prior to each test and the

`teardown()` method is called after. Both are optional and either may be omitted independently. Other methods and variables may also be declared inside a group and will be available to all tests of that group.

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```