Article
Browse Code
View Stats
Revisions (24)
Comments (3)
Posted 7 Nov 2015

Tagged as
Embedded
C++
C
VS2012
VS2013

Stats
44.2K views
1.2K downloads
14 bookmarked

Using GoogleTest and GoogleMock frameworks for embedded C

Michael Pan

9 Nov 2015 CPOL

Rate me!

Presenting techniques for effective usage of Google unit test frameworks in embedded environment.

Download UnitTestForEmbeddedC.zip - 17.6 KB

Introduction

This article is about unit testing. Although unit testing is quite popular in various programming languages , t programming is usually overlooked. This is ironic because embedded programmer may benefit from unit tes write for embedded systems then you know how hard it is to debug your code. You add logs, connect JTAG you curse, and, if so, this article was written to help you. Give it a try.

In this article I use GoogletTest and GoogleMock frameworks to write unit tests. I assume that reader is familiar with these frameworks. Otherwise you should read Google test primer and Google mock for dummies first. While the GoogleTest could be easily adapted for C functions testing (you will see how in next section), the GoogleMock has a little to propose to C programmer. The GoogleMock was designed for mocking C++ interfaces and it relies on virtual functions which are lacking in C language. Of cause, without mocking interfaces, the unit testing becomes very limited. In this article I suggest two different approaches for C functions mocking and then bundle them into one solution that addresses majority of unit testing scenarios. This article also covers some design aspects to help you in unit tests writing.

go to top

DISCLAIMER: The techniques that I describe in this article are not new. Some of them were borrowed from "Test driven development for embedded C" book. The book is an excellent unit testing resource for C programmer but, in my opinion, it should have described in more details how to utilize frameworks for C functions mocking. There are a lot of articles and frameworks over the Internet trying to address the C mocking problem, but I found them incomplete for my needs. Most of them are limited to GNU/Linux environment and provide no practical guidance for unit testing writing. Also, in this article, I propose HOOK mocking technique that I haven't found in any other framework. Anyway, I do not claim for a best solution and it is up to you to decide what is good for your code.

Testing C functions with GoogleTest

Let's say we have simple queue code and we want to put it under test. The code resides in *Queue.c* file.

Hide Copy Code

```
#define QUEUE_SIZE 15

static int queue[QUEUE_SIZE];
static int head, tail, count;

void QueueReset()
{
    head = tail = count = 0;
}

int QueueCount()
{
    return count;
}

int QueuePush(int i)
{
    if (count == QUEUE_SIZE)
        return -1;
    queue[tail] = i;
    tail = (tail + 1) % QUEUE_SIZE;
    count++;
    return 0;
}
...
```

The trick is simple, after creating C++ file for the test, include the C file into it and the C code will become accessible to the testing code. Below is the code of *QueueUnitTest.cpp* file.

Hide Copy Code

```
#include "Queue.c"

TEST(QueueUnitTest, ResetTest)
{
    EXPECT_EQ(0, QueuePush(1));
    EXPECT_EQ(1, QueueCount());
    QueueReset();
    EXPECT_EQ(0, QueueCount());
}

TEST(QueueUnitTest, CountTest)
{
    for (auto i = 0; i < QUEUE_SIZE; i++)
        EXPECT_EQ(0, QueuePush(i));
    EXPECT_EQ(QUEUE_SIZE, QueueCount());
}
...
```

You may ask why not to include *Queue.h* instead of *Queue.c*? Indeed, this will allow you to call the C functions, but you won't get access to static functions and static global variables declared in the C file.

Since we have mentioned global variables, let's discuss them in more details. There are no classes in C, but in well designed C program, the code is usually divided into modules which reside in separated files. Thus the C module could be considered as C++ class equivalent. If so then the module global variables are class members and you should be able to initialize them before test running. Consider adding `ModuleName_InitGlobals` method and put global variables initialization into it (this is module constructor). Here is example of such initialization in *Queue.c* file.

Hide Copy Code

```
void Queue_InitGlobals()
{
    head = tail = count = 0;
}

void QueueReset()
{
    Queue_InitGlobals();
}
...
```

The test fixture class can now call `Queue_InitGlobals` function to prepare module for test.

Hide Copy Code

```
class QueueUnitTest : public ::testing::Test
{
public:
    QueueUnitTest()
    {
        Queue_InitGlobals();
```

Posted 7 Nov 2015


```
    }
};

TEST_F(QueueUnitTest, ResetTest)
{
    EXPECT_EQ(0, QueuePush(1));
    EXPECT_EQ(1, QueueCount());
    QueueReset();
    EXPECT_EQ(0, QueueCount());
}
...
```

Simple, isn't it?

Mocks in C code

In previous section we saw how to test C functions, but what if these functions call other functions which, in turn, do not reside in the module we test? For example, there could be OS log service that our queue should call on every operation or it should signal GPIO if queue is full. To handle these situations we need to create mock objects and teach the code to call mocks instead of production code. I found two useful techniques for C functions mocking and I suggest choosing between them based on the nature of mocked functions.

Mocking services

The first technique proposes to cut functions off your code and replace them with fake implementations. These implementations, in turn, will call to mock objects and, thus, allow using mock functionality. This approach is mostly appropriate for hardware services like timers and GPIO and also for OS entities like logs and semaphores. Indeed, such services are not related to your application code and should not be put under test, so their fake implementations provide your application with kind of virtual runtime environment.

Let's say we have an oscillator and GPIO registers and we wrap them with the following functions.

Hide Copy Code

```
int GetTime()
{
    // Code that reads oscillator register
    ...
}

int ReadGpio(int line, int *value)
{
    // Code that reads from GPIO line
    ...
}

int WriteGpio(int line, int value)
{
    // Code that writes to GPIO line
    ...
}
```

Go to top

The functions should be reimplemented in our testing code. The new implementations will call to the corresponding mock objects.

Hide Copy Code

```
int GetTime()
{
    return TestFixture::_oscillator->GetTime();
}

int WriteGpio(int line, int value)
{
    return TestFixture::_gpio->WriteGpio(line, value);
}

int ReadGpio(int line, int *value)
{
    return TestFixture::_gpio->ReadGpio(line, value);
}
```

In the code above the mock objects are retrieved from static members of `TextFixture` class. You will find a detailed explanation of `TextFixture` stuff in further sections.

Below is an example of code that calls to the services. The code resides in *CallingServices.c* file.

Hide Copy Code

```
void ShortCircuit(int timeout)
{
    int startTime = GetTime();
    while(GetTime() - startTime < timeout)
    {
        int signal;
        ReadGpio(0, &signal);
        WriteGpio(1, signal);
    }
}
```

The unit test for the ShortCircuit function is shown below. Again, the expectations are set on mocks stored in the `TextFixture` class.

Hide Copy Code

```
#include "Fixture.h"
#include "CallingServices.c"

namespace EmbeddedCUnitTest {

class CallingServicesUnitTest : public TestFixture
{
public:
    // The fixture holds pointers to mock objects in _oscillator and _gpio members
    ...
};

TEST_F(CallingServicesUnitTest, ShortcutTest)
{
    EXPECT_CALL(*_oscillator, GetTime()).Times(3)
                .WillOnce(Return(10)).WillOnce(Return(100)).WillOnce(Return(111));
    EXPECT_CALL(*_gpio, ReadGpio(0, _)).Times(1)
                .WillOnce(DoAll(SetArgPointee<1>(1), Return(0)));
    EXPECT_CALL(*_gpio, WriteGpio(1, 1)).Times(1);

    int timeout = 100;
    ShortCircuit(timeout);
}
```

Mocking C modules

The second mocking technique is more suitable for mocking your own code. Suppose there is a module that uses our queue module. Let's say the code is triggered by a packet received from network and the packet content is pushed into the queue.

Hide Copy Code

```
int OnDataReceived(int data)
{
    if (QueuePush(data) == -1)
    {
        return -1;
    }
    return 0;
}
```

Our goal is to put expectations on queue module functions. For this purpose we will add, at the beginning of the functions, a call to its mock object. To do so we will use macro magic. I know that as a C programmer you love macro magic! ...but first we need to write queue mock class.

Hide Copy Code

```
class QueueMock
{
public:
    MOCK_METHOD0(QueueReset, void());
```

```
    MOCK_METHOD0(QueueCount, int());
    MOCK_METHOD1(QueuePush, int(int));
    MOCK_METHOD1(QueuePop, int(int*));
};
```

The macro that we are going to put into queue code calls to mock in testing code and does nothing in production code. Let's call these macros **HOOK**s.

Hide Copy Code

```
#if defined(__cplusplus)

#define MOCK_HOOK_P0(f) {return TestFixture::GetMock<MOCK_CLASS>().##f();}
#define MOCK_HOOK_P1(f, p1) {return TestFixture::GetMock<MOCK_CLASS>().##f(p1);}
#define MOCK_HOOK_P2(f, p1, p2) {return TestFixture::GetMock<MOCK_CLASS>().##f(p1, p2);}
...
#else

#define MOCK_HOOK_P0(f)
#define MOCK_HOOK_P1(f, p1)
#define MOCK_HOOK_P2(f, p1, p2)
...
#endif
```

The `TestFixture::GetMock<MOCK_CLASS>()` returns a reference to mock object of type `MOCK_CLASS`. The `MOCK_CLASS` is a placeholder for mock class and it is defined with `#define` directive. We will discuss how exactly it works in the next section.

Here is the code of *Queue.c* with HOOK macros in it.

Hide Shrink Copy Code

```
#include "MockHooks.h"
...

void QueueReset()
{
    MOCK_HOOK_P0(QueueReset);
    Queue_InitGlobals();
}

int QueueCount()
{
    MOCK_HOOK_P0(QueueCount);
    return count;
}

int QueuePush(int i)
{
    MOCK_HOOK_P1(QueuePush, i);
    ...
    return 0;
}

int QueuePop(int *i)
{
    MOCK_HOOK_P1(QueuePop, i);
    ...
    return 0;
}
```

Go to top

Finally, we can write expectations in our tests (*CallingModuleWithHooksUnitTest.cpp* file).

Hide Copy Code

```
#include "Fixture.h"
#include "QueueMock.h"
#include "CallingModuleWithHooks.c"

class CallingModuleWithHooksUnitTest : public TestFixture
{
public:
    CallingModuleWithHooksUnitTest() : TestFixture(new QueueMock)
    {
        CallingModuleWithHooks_InitGlobals();
    }
};

TEST_F(CallingModuleWithHooksUnitTest, OnDataReceivedPositiveTest)
{
    EXPECT_CALL(GetMock<QueueMock>(), QueuePush(1)).Times(1).WillRepeatedly(Return(0));
    EXPECT_EQ(0, OnDataReceived(1));
}

TEST_F(CallingModuleWithHooksUnitTest, OnDataReceivedNegativeTest)
{
    EXPECT_CALL(GetMock<QueueMock>(), QueuePush(1)).Times(1).WillRepeatedly(Return(-1));
    EXPECT_CALL(*_gpio, WriteGpio(10, 1)).Times(1);
    EXPECT_EQ(-1, OnDataReceived(1));
}
...
```

Well, how do you like it? You don't?! I can see your loaded guns are aimed at me because I did not do it by the book. Indeed, mock functions should not be hooked into production code but should substitute it. All true, but HOOKs have several advantages. Let me compare HOOKs and "fake functions" approaches.

1. With HOOKs you don’t need to maintain fake implementations which usually reside in separate files. Anytime you change function signature, you can update its HOOK right away.
2. You can configure HOOKs of a given mock to continue execution of C function code after its hook has been hit. This way you may create more complex scenarios which are sort of integration tests (I do not discuss this feature in the article but you can find it in the attached code). Doing it with fake functions is tedious.
3. The HOOKs appoach allows you to grow your tests by adding new module tests into the same project. As a result, your testing and production code projects will look quite the same. With fake functions you will have multiple test projects with only few modules tested in each of them.
4. Assuming that every module in your code has its own mock class, you will get compilation error if some function is moved from one compilation unit to another, but you forget to update the mock class. This is because of `MOCK_CLASS` macro. In case of fake functions, the testing code will continue working, but the mock class will stop reflecting the real code design.
5. The biggest disadvantage of HOOKs is that you can forget to add HOOK in some function. Then two things may happen:
   a. Your mock expectation will not be satisfied and you will find out that HOOK is missing.
   b. Your test will keep calling production code. This is not a problem as long as your test is not affected by the call and this, I believe, will be true for most of the cases.
   Anyway, you must acquire some discipline and do not miss HOOKs, but wait... you are embedded C programmer and if you have not yet learned to be highly organized, you won't keep your job for long.

Another problem to address is module's global variables. I have already showed, in first section, how to initialize global variables in test fixture class. Using C modules mocks introduces a new challenge: the test must initialize global variables of the mocked modules as well. Again, assuming we have made a right design decision of having dedicated mock class for every module, we can put a call to module's initialization function in mock constructor.

Hide Copy Code

```
class QueueMock : public ModuleMock
{
public:
    QueueMock()
    {
        Queue_InitGlobals();
    }
    ...
}
```

This approach for global variables initialization is a huge advantage of HOOKs technique. In case of fake functions the global variables must be redefined in testing code. Indeed, if your module accesses another module's global variables, the code won't compile till you add the variables to the testing code. The HOOKs technique solves the problem by bringing the original module into the project.

Creating test fixture

By now we have two kinds of mocks: services and modules mocks. In this section we will create test fixture that will ease accessing these mocks. We will start with services mocks. Since services mocks wrap hardware and OS, it is logical to store them in static objects. The mocks could be created in our fixture class as following.

Hide Copy Code

```
class TestFixture : public ::testing::Test
{
```

```
public:
    TestFixture()
    {
        _oscillator.reset(new ::testing::NiceMock<OscillatorService>());
        _gpio.reset(new ::testing::NiceMock<GpioService>());
    }

    ~TestFixture()
    {
        _oscillator.reset();
        _gpio.reset();
    }

    // Services
    static std::unique_ptr<OscillatorService> _oscillator;
    static std::unique_ptr<GpioService> _gpio;
};
```

The `TestFixture` class is designed to be base class for all your test fixture classes. If some test requires different mock for a service, the test fixture class can reinitialize the mock pointer in its constructor.

The destructor resets the pointers to ensure expectations check (mock expectations are checked in mock destructor). Without it, the static objects will keep holding the mocks and the mocks won't be destructed.

Now let's talk about HOOKs. Given mock class and corresponding C module, we need an ability to find and call mock object from the HOOK macro. This is done in two steps. First, we define a base class for all our module mocks.

Hide Copy Code

```
class ModuleMock
{
    virtual ~ModuleMock() {}
};

class QueueMock : public ModuleMock
{
    ...
}
```

Second, we store mocks in `TestFixture` static member, so the macros can access them through static method.

Hide Shrink Copy Code

```
class TestFixture : public ::testing::Test
{
public:
    TestFixture(ModuleMock *mocks)
    {
        _modulesMocks.reset(mocks);
        ...
    }

    ~TestFixture()
    {
        _modulesMocks.reset();
        ...
    }

    template<typename T>
    static T& GetMock()
    {
        auto ptr = dynamic_cast<T*>(_modulesMocks.get());
        if (ptr == nullptr)
        {
            throw std::exception();
        }
        return *ptr;
    }

    // Modules mocks
    static std::unique_ptr<ModuleMock> _modulesMocks;
    ...
};
```

Go to top

Every test fixture class inherits from `TestFixture` and passes its mocks to the `TestFixture` constructor. Here is an example of mocks creation in already familiar *CallingModuleWithHooksUnitTest.cpp* file.

Hide Copy Code

```
class Mock
    : public ::testing::StrictMock<QueueMock>
    , public ... // another mock
    , ... // more mocks
{
};

class CallingModuleWithHooksUnitTest : public TestFixture
{
public:
    CallingModuleWithHooksUnitTest() : TestFixture(new Mock)
    {
        ...
    }
};

TEST_F(CallingModuleWithHooksUnitTest, OnDataReceivedPositiveTest)
{
    EXPECT_CALL(GetMock<QueueMock>(), QueuePush(1)).Times(1).WillRepeatedly(Return(0));
    EXPECT_EQ(0, OnDataReceived(1));
}
...
```

As you can see in the `TestFixture` code, the `GetMock` method allows to retrieve mock object of a given type. In HOOK macro, the `GetMock` method uses the `MOCK_CLASS` as template parameter. The `MOCK_CLASS` macro have to be defined before including the C file into your test. For example, the *QueueUnitTest.cpp* file should contain the following code.

Hide Copy Code

```
#define MOCK_CLASS QueueMock
#include "Queue.c"
```

Pay attention that if multiple tests in the project refer to the queue mock object, all of them will use the same `QueueMock` class. This is because the `MOCK_CLASS` for *Queue.c* is solely defined in *QueueUnitTest.cpp*. If you want to use a different queue mock class for specific test, you will need to inherit it from already existing `QueueMock` class.

So now we have one fixture that has it all. It contains both services and modules mocks. As you may notice, all mock objects reside in static variables. This would have never worked in C++, but in C, only free functions are mocked and every module needs only one mock instance in every given test.

Growing your tests

As I have mentioned earlier, the HOOKs approach allows you adding more and more C module tests into the project. This is quite convenient, but you don't need to put all the embedded code under test at the first day. Once you choose a module to test, track its dependencies, create stub implementations for the dependent modules and put HOOKs into them. Here is an example of such *stub.c* file content.

Hide Copy Code

```
#include "MockHooks.h"

void Foo()
{
    MOCK_HOOK_P0(Foo);
}
```

Then create mock class and add an empty unit test for the stub implementation. Below is the content of *StubUnitTest.cpp* file.

Hide Copy Code

```
#include "StubMock.h"

#define MOCK_CLASS StubMock
#include "Stub.c"
```

Once you decide to put real module under test, all you need to do is to replace the C file in the code above with the real module file.

Code sample

In the attached code you will find some goodies that I don't describe in this article. So I advise you to read the code before applying the learned techniques on your own project. Of cause, start reading with the README file.

Summary

I hope this article helped you to understand how to utilize Google test and Google mock frameworks and write unit tests for your embedded C code. The described techniques are not hard to understand but they require from you pedantic attitude and good understanding of your software design. Try to apply these techniques to some pieces of your code and you will be surprised by how many stupid bugs you will find ...and most important, it will happen before running the code on your target.

License

Share

About the Author

Michael Pan
Software Developer
Israel

Follow this Member

No Biography provided

Go to top

Comments and Discussions

Add a Comment or Question

Email Alerts

Search Comments

Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

Same namespaces — Member 14029312 — 24-Oct-18 23:01

Having trouble testing my c code — ecat87 — 1-Nov-17 15:30

Migrating this project to MSVC 2005 — nagarjunr — 27-Jan-16 2:15

Last Visit: 15-Mar-20 21:35 Last Update: 15-Mar-20 21:37 Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.