

# Test Driven Development for Embedded Software

James Grenning

San Jose, April 2007, ESC Class# 241

Did you write any code this week? Raise your hand. Do you know if your code works? If you are like the hundreds of people I have asked that question to, your hand is probably down right now. What about those of you with your hand still up, how do you know? My guess is that you either have comprehensive automated unit tests for your code, or you are deluding yourself. Are you confident that you did not introduce any side effect defects in the code written this week? With automated tests you could be right. Without the automated tests you can't be so confident.

Embedded systems expert Jack Ganssle says "The only reasonable way to build an embedded system is to start integrating today... The biggest schedule killers are unknowns; only testing and running code and hardware will reveal the existence of these unknowns."<sup>[GANSSE]</sup> Jack goes on to say that "Test and integration are no longer individual milestones; they are the very fabric of development." This paper is about a way to weave test and integration into the fabric of development. It's called Test Driven Development.

Test Driven Development is a practice that concurrently develops automated unit and acceptance tests and the working code that satisfies those tests. This technique was developed in the Smalltalk community by Ward Cunningham, Kent Beck and others. Can embedded developers successfully adopt this practice for embedded software? If you ask me the answer is yes. This paper describes how to use TDD for embedded software development. TDD is often used in Agile development, please see my other paper on Agile Development.<sup>[GREN-AGL]</sup>

## Development and Execution Environments

Embedded software development brings along a few additional challenges. The development environment usually differs from the target execution environment. Development systems are usually standard off the shelf products. Target systems are custom, limited in availability, expensive and usually shared by the development team members. As you know, the target is often new and untried hardware, possibly still in the prototype shop.

I've seen hardware prototypes costing over \$1 million. Not every software engineer will get one of these babies. This means sharing; and sharing means waiting. Waiting kills productivity. Even with access to target hardware, development time is slowed whenever we test on the target. Downloading and running in the target takes time, and it's a difficult and expensive environment to debug in.

Testing in the target is necessary, but not always possible or practical. Fortunately, there is an alternative to executing in the target. I run most of my code in my development

environment using automated unit and acceptance tests. The cost for doing this is low and I think it is paid for many times over as compared to testing strictly in the target.

## Test Driven Development Cycle

Test Driven Development is a state-of-the-art software development technique that results in very high test coverage and a modular design. Kent Beck, author of *Test-Drive Development by Example*<sup>[BECK]</sup> describes the TDD cycle as:

1. Quickly add a test
2. Run all the tests and see the new one fail
3. Make a little change
4. Run all the tests and see the new one pass
5. Refactor to remove duplication

This cycle is designed to take only a few minutes. Code is added incrementally. Every few minutes you find out if the code you just wrote is doing what you thought it would do. The tests are automated. All tests are re-run with every change. Tests are a reflection of the code, so tests serve as executable documentation. They are examples of how to use the code.

Automated tests and TDD are really important for finding side effect defects. You know those annoying little bugs that you chase for days on end long after the code was developed just because you made a bad assumption a couple months ago. When practicing TDD this is not uncommon

- Add a new test.
- Figure out how to make it pass.
- Add the code; make that modification based on your best information and knowledge.
- Run the tests.
- The new test passes
- But a few previously passing tests now fail.

What happened? You violated some earlier assumption. Fortunately, instead of keeping that assumption in your head, or in some document, it was *coded* in your tests. So the code keeps you from violating your previous assumptions. The code tells you when it is broken.

Without the automated tests, you have a good chance of injecting a bug with any change. That bug might be found later in some test activity if you are lucky. If you aren't lucky, the bug will be found by your customer and do damage in a live environment. The bug will be more difficult to find because it will have been days, weeks, or months since it was injected. Having the code tell you when it is broken is very powerful, and can save a lot of time.

With well designed automated tests you have a good chance of finding the problem quickly. You will have cause and effect isolated. Back out the change and see the test

pass, then put the change back and see it fail. Then you must find a change that keeps all tests passing. One thing that is not tolerated is broken unit tests. New behavior is not added unless it does not break existing behavior.

This cycle is very powerful. It's a feedback loop. Every change is like an experiment. Make a change and then confirm behavior. It has saved me and others many hours and days of debug time; more than paying for the investment in test code.

TDD forces each module to be testable in isolation. This isolation means that the design must be modular and loosely coupled. Loose coupling is a desirable quality of software systems, as it makes them easier to understand and change. In Object Oriented languages loose coupling is achieved by using Interfaces to decouple the parts of the system from each other. If you are working in C you can adopt conventions that mimic the interfaces of OO, although you'll get very little help from the language.

### ***TDD and Embedded***

I've heard that TDD can't be used for embedded. Reasons cited are

- We have hardware dependencies
- We have real time constraints
- We have limited memory
- We can't use an OO language

Despite these complaints the rapid feedback cycle of TDD can, in fact, be used by embedded developers. I know because my colleagues and I are doing it. The issues above do present some unique problems; but there are creative solutions that resolve them. To use TDD you must develop a testable design and hardware dependencies must be managed. The design must be structured so that most of the system's modules can be run in both the target, and the development system.

Embedded developers find themselves in these situations:

- There is no target hardware, its under development.
- There is a prototype, but the hardware guys are using it and the only available slot is next Friday at 10:45 PM.
- There's an evaluation board, with none of the specialized IO.
- The final hardware becomes available just as the product is due to ship next week.

Is it safe to say we have some challenges? I'd say so. However, embedded developers can mitigate the risk of limited access to hardware by designing the code so that it runs in multiple execution environments. We have to be able to run our tests in the development platform so that when hardware is finally available to us, we can verify operation, and hopefully not have many remaining problems to find. We get a bonus from this too, platform independence; a good thing even if we were not doing TDD.

## What do these tests look like?

Consider a home security system named HOME GUARD. One of the unit tests in the system is designed to assure that the system is in the right state after power on initialization. This test is concerned with the state of the front panel object and the phone object. If initialization is meeting its requirements the system will not be generating any audible or visual alarms at power up. It won't call the police, it will have the word "READY" in the front panel display.

Running this test with the actual target hardware would require a manual step, or some pretty expensive instrumentation. To avoid this in the example below, the panel and the phone are test stubs, also known as Mock Objects.<sup>[MACKINNON]</sup> The Mock Objects stand in for software that knows how to manipulate the hardware. The stand-ins have no knowledge of the hardware. They just implement the same interface as the hardware dependant code and remember what they have been told to do so our test can verify they were given the right instructions.

This test is written in C++ using CppTestTools<sup>[CPPTEST]</sup> Not shown is the system setup that the test harness controls in `SetUp` and `TearDown` functions. [In `SetUp` the system is initialized, associating the test stubs with the core of the security system.](#)

```
TEST (HomeGuard, PowerUp)
{
    CHECK (false == panel->isArmed());
    CHECK (false == panel->isAudibleAlarmOn());
    CHECK (false == panel->isVisualAlarmOn());
    CHECK ("READY" == panel->getDisplayString());
    CHECK (true == phone->isPhoneIdle());
}
```

Tests have to pass or fail. There is no almost. They are a series of Boolean checks. A project may be made up of hundreds or thousands of tests like this simple example. Don't let those numbers scare you. The tests get written one at a time. When a test is written, the code is written to make that test pass, hence the name Test Driven Development. Code is developed incrementally. All the time making sure the automated tests pass.

When the unit tests are run with passing tests you get output that looks like this

```
OK (79 tests, 78 ran, 123 checks, 1 ignored, 0 filtered
out)
```

If there is a failure you get pointed right to the failing test like this

```

Failure in TEST(HomeGuard, ConfigurationDownload)
  HomeGuardTest.cpp(64)
    expected <COMMUNICATION PROBLEM EXISTS>
    but was <COMMUNICATION PROBLEM EXITS>

```

Here is what the same test might look like when programmed in C.

```

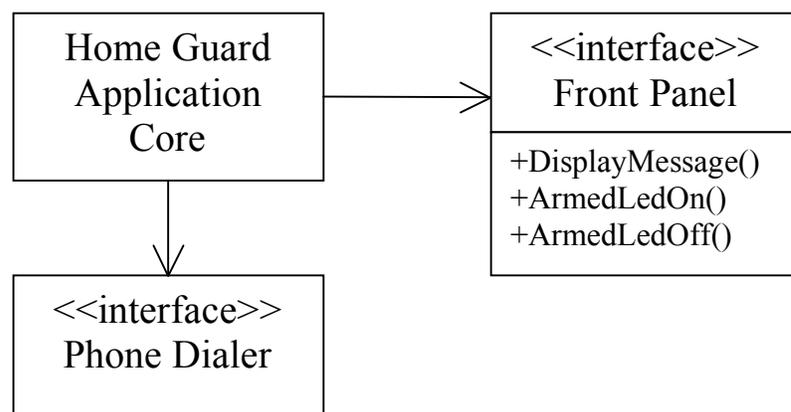
TEST(HomeGuard, PowerUp)
{
    CHECK(false == mockPanel_isArmed());
    CHECK(false == mockPanel_isAudibleAlarmOn());
    CHECK(false == mockPanel_isVisualAlarmOn());
    CHECK("READY" == mockPanel_getDisplayString());
    CHECK(true == mockPhone_isPhoneIdle());
}

```

Note that this suggests structuring C code in an Object Oriented way. The interface to the panel and phone are defined in header files. The real panel implements those functions, as does a mock version of it. When creating the test executable you link in the mock versions, when linking for production you link in the real implementations.

### ***What does the design look like?***

The core of Home Guard talks to the hardware through the two interfaces. In C++ an interface is defined as a class with pure virtual functions. In C an interfaces is represented in a header file, with function prototypes for interface functions. For testing mock implementations are used. The real implementations are used for a full product build. In C++ the linker can be used or dynamic binding to select mock vs. real implementations. In C we would most likely use the linker to select the desired implementation.



This design improves the Home Guard software platform independence. Tests run on the development system or the target system. The hardware dependencies have to be isolated through interfaces with stubs, or faked-out implementations used for testing. That topic is covered in detail in another paper “Progress before hardware”<sup>[GRENNING]</sup>.

## ***Embedded TDD Cycle***

Executing tests on the development system is both a risk and a risk reduction strategy. It is risk reduction, because we are proving the behavior of the code prior to execution in the target. It is a risk because the target and the development platform compilers may have different capabilities, implementations, and bugs.

Prior to target availability we risk using compiler features that are not supported by the target compiler. To mitigate this risk we add another step to the embedded TDD cycle: periodically compile with the target’s cross-compiler. This will tell us if we are marching down a path towards porting problems. What does periodically mean? Code written without being compiled by the target’s compiler is at risk of not compiling on the target. How much work are you willing to risk? A target cross-compile should be done before any check in, and probably whenever you try out some language feature you have not used before. Your team should have an agreed upon convention. It would be a shame to use some C++ feature only to find that the target compiler does not support it.

Once target hardware is available, we’ll continue to use the development systems as our first stop for testing. We get feedback more quickly and have a friendlier debug environment. But, testing in the development environment introduces another risk: execution may differ between platforms. To lessen this risk we’ll periodically run the unit tests in the prototype. This assures that the generated code for both platforms behaves the same. Ideally the tests are run prior to check-in. You should consider how much of your work is being risked by not getting the feedback from running tests in the target. Keep in mind that if you have limited memory in your target, tests may have to be run in batches.

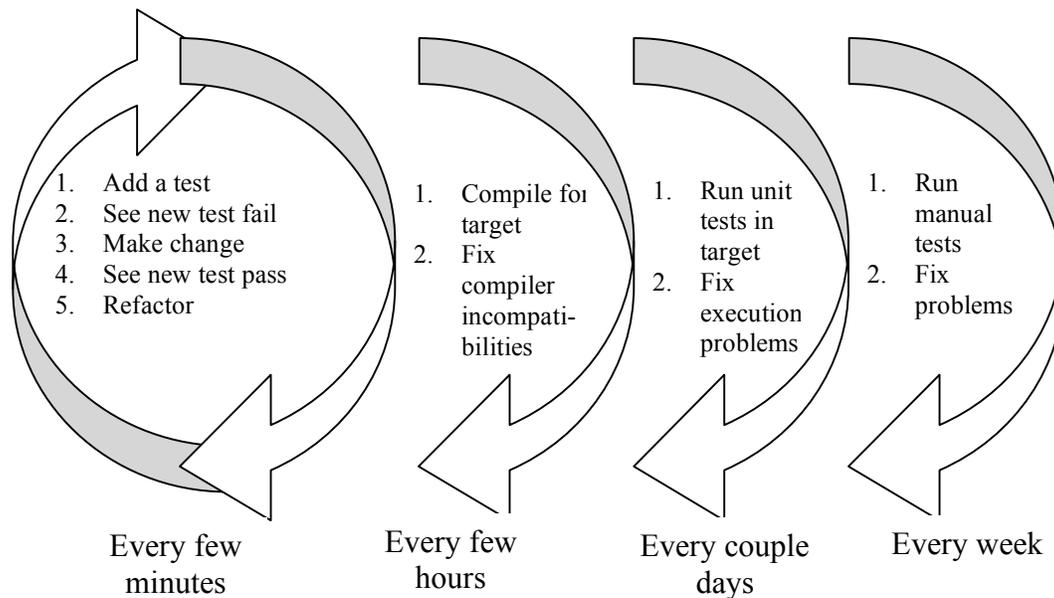
With target hardware available you should add tests for the hardware and/or tests for code that uses the hardware. Write tests that show how you expect to use the hardware and make sure it works as you expect. The hardware engineers might just thank you.

Automated unit tests are more difficult to create when the hardware interacts with the real world. The tests may involve external instrumentation or manual verification. Consider a test for turning on an LED. You can write a test to turn on the LED, but you need someone’s eyes, or some electrical or optical sensor to see if the LED is on. The later two options sound expensive. The former is time consuming. To minimize the manual tests, the hardware layer is made as thin as possible. We may never achieve 100% test automation; so we will strive to fully test the core of the application and manually test the final connections to the hardware.

With a fully functional target platform we can do end-to-end testing. Ideally the end-to-end testing would be automated, but this is difficult and expensive to achieve in most cases. One big challenge in end-to-end testing is running the system through all the scenarios it has to support. Uncommon scenarios still have to work, so how do we get the system into a particular state and have the right triggering event to occur at exactly the right time? It turns out that controlling the system state and triggering certain events is easier in our test environments. Our mocked-out hardware implementations can be instructed to give the responses needed to exercise the obscure paths through the code.

For example in the home security system, what if the phone dialer cannot get dial tone? How many retries do we attempt? How long does the system delay between retries? What if some alarm system models have the second phone line option, when do we call on the backup phone line? This sounds like an involved manual test. By using a mock phone dialer implementation this behavior can be easily tested. A dial tone failure can be faked and the system response can be monitored and confirmed.

## Summary



This diagram represents the embedded TDD cycle. It is part of a quality improvement and risk reduction strategy to keep bugs out of the software, and helps keep the team from getting stalled.

Manual test are too expensive to run frequently. Any change can introduce a killer bug. One problem with manual tests is that they have to be consistently set up and executed. They are repetitious and boring. They are unlikely to be run often enough. If you don't run them they cannot find problems. When they do find bugs it is often months after the bug was injected, so the time to fix the problem increases. Therefore we want to

automate as many tests as possible. Automation has a cost, but I have found that writing automated tests is cheaper than not writing tests. The tests easily pay for themselves in less debug time.

Using this Embedded Test Driven Development Cycle can provide embedded software engineers a valuable test bed for their software. Automated unit tests tell the programmer when their code is broken. Unit tests provide an example of how the code works and is therefore a valuable form of documentation. The tests are the first place to look when trying to understand how a given module works.

Unit tests are not an end in themselves. The system has to work and be delivered. Automated acceptance tests, load tests and timing tests are likely necessary, but a topic for another day.

## Appendix A – Example TDD Session in C++

In this example we are going to develop something familiar to you all, a Stack using TDD. The objective of this is to give you the feel of TDD.

All C++ TDD sessions start with three files. CppTestTools will generate the initial starting point. I used this CppTestTools command in the directory with my source code and makefile.

```
NewClass Stack
```

This creates three files: Stack.h, Stack.cpp, and StackTest.cpp. The script also updated the makefile in this directory.

We start with a very defensive class declaration, trying to prevent the C++ compiler from *helping* us too much.

The constructor is `explicit` to prevent conversion automatic conversion if we were to change the `Stack()` signature to include a single parameter. If we want conversion, we will write tests for it and remove `explicit`.

```
//Stack.cpp
#include "Stack.h"

Stack::Stack()
{
}

Stack::~Stack()
{
}
```

The destructor is `virtual`. If you are sure you won't inherit from this class, and you are really worried about the space, you can delete that keyword.

```
//Stack.h
#ifndef D_Stack_H
#define D_Stack_H

class Stack
{
public:
    explicit Stack();
    virtual ~Stack();

private:
    Stack(const Stack&);
    Stack& operator=(const Stack&);

};
#endif // D_Stack_H
```

The copy constructor and assignment operator are private and not implemented to prevent the compiler from generating those items for us. By hiding these, we essentially prohibit passing and returning objects by value, an easy and wasteful thing to accidentally do.

```
//StackTest.cpp
#include "UnitTestHarness/TestHarness.h"
#include "Helpers/SimpleStringExtensions.h"
#include "Stack.h"

// to make sure this file gets linked in to your test main
EXPORT_TEST_GROUP(Stack);

namespace
```

Starter versions of the constructor and destructor are provided. Note they are always in the cpp file and not in the header. Event though these appear to be empty, there is compiler generated code between those curly braces. So we put it in the cpp file by default to keep the footprint small. If we were not talking embedded here, we might not care as much.

Here is the initial failing tests case. When doing TDD, you only write production code when there is a failing test case. The TEST macro defines a test case. SetUp() is run before each TEST invocation, and TearDown() is run after each TEST's closing curly brace. This makes sure that our tests don't talk to each other and remain independent.

```
Failure in TEST(Stack, Create)
  StackTest.cpp(24)
  Start here

.....
Errors (1 failures, 1 tests, 1 ran, 0 checks, 0 ignored, 0 filtered out)
```

Now that we have the boiler plate out of the way, we can get down to work. A good place to start is to make a list of the tests cases to write, and then order those tests such that we can start with the easier cases and work our way to the more difficult cases. This lets us examine the interface first and then work out the implementation details when we have a good idea of how to interact with the object.

Test case brainstorm (and initial guess at ordering)

- Empty stack
- Not empty
- Push one item get it back with a Pop
- Mix up Pushes and Pops to exercise the stack
- Specify the stack size
- Full stack
- Push to a full stack
- Pop from an empty Stack

We may not know what to do about some of these test cases, for example what should we do with a stack that has blown its top, or a programming error that results in popping an empty stack. We can start on the parts we understand and get answers for the other parts later.

When a stack is created, it is empty, here is the test that guarantees that.

```
TEST(Stack, Empty)
{
    CHECK(stack->IsEmpty());
}
```

This test fails, it does not even compile, so we add just enough code to get the test to compile, then link, and fail.

```
//add this to the header file
bool IsEmpty();

//add this to the cpp file
bool Stack::IsEmpty()
{
    return false;
}
```

The failing test demonstrates that our new test is successfully installed in the test framework.

```
Failure in TEST(Stack, Empty)
  StackTest.cpp(28)
  CHECK(stack->IsEmpty()) failed
.
Errors (1 failures, 2 tests, 2 ran, 1 checks, 0 ignored, 0 filtered out)
```

Using the “fake it ‘til you make it” technique, change the false to true and see the test pass. <sup>[BECK]</sup>

```
OK (2 tests, 2 ran, 1 checks, 0 ignored, 0 filtered out)
```

Don’t let the faking it bother you. It was a very easy way to get the tests to pass, and we will write another test for the other side of the condition, not empty, and then we will have eliminate that “fake it” code. We continue with the flow: write a test, make it compile, make it link, make it fail and finally make it pass.

The test:

```
TEST(Stack, NotEmpty)
{
    stack->Push(12);
    CHECK(!stack->IsEmpty());
}
```

The header file changes:

```
class Stack
{
public:
    explicit Stack();
    virtual ~Stack();

    bool IsEmpty();
    void Push(int);

private:
    int index;

    Stack(const Stack&);
    Stack& operator=(const Stack&);

};
```

The cpp file changes:

```
Stack::Stack()
: index(0)
{
}

bool Stack::IsEmpty()
{
    return index == 0;
}

void Stack::Push(int value)
{
    index++;
}
```

The test result:

```
...
OK (3 tests, 3 ran, 2 checks, 0 ignored, 0 filtered out)
```

You can see the fake went away and the real implementation took its place. It is still incomplete, but evolving and converging on the solution. You might think these tests are too trivial, and I would agree with you, except that more complexity is coming. The simple things as well as the complex things must work, we never know where we will insert a bug, so we continue to follow the cycle. We prevent a whole batch of simple

mistakes that may be very difficult to find when integrated with hundreds or thousands of lines of other code.

Now let's get the Stack working for a single entry. This way we get the interface designed first, make sure the stack works for that simple case and have less code to write to get the more general/strenuous case to work. Here are our results.

```
TEST(Stack, PopReturnsWhatWasPushed)
{
    stack->Push(12);
    LONGS_EQUAL(12, stack->Pop());
}

TEST(Stack, PushOnePopOneEmpty)
{
    stack->Push(12);
    stack->Pop();
    CHECK(stack->IsEmpty());
}
```

The PushOnePopOneEmpty test was an after-thought; after we pop a stack with one item in it the stack should be empty, so we added that test case. That check could have gone in the PopReturnsWhatWasPushed test, but it is better to keep the test cases short and to the point.

The header file changes:

```
//Added
int Pop();
```

The cpp file changes:

```
void Stack::Push(int value)
{
    index++;
}

int Stack::Pop()
{
    index--;
    return 12;
}
```

There's that "fake it" in there again this time returning a hard-coded 12. Now we are ready to store values into the stack, I guarantee the fake it will be gone soon. We fake it to get feedback quickly. This also helps us to write more comprehensive tests

The test result:

```
.....  
OK (5 tests, 5 ran, 4 checks, 0 ignored, 0 filtered out)
```

Finally we'll add the guts of the stack, but not before writing a test that exposes the current implementation's weakness.

```
TEST(Stack, PushPopGeneralCase)  
{  
    stack->Push(10);  
    stack->Push(11);  
    stack->Push(12);  
    LONGS_EQUAL(12, stack->Pop());  
    CHECK(!stack->IsEmpty());  
    LONGS_EQUAL(11, stack->Pop());  
    CHECK(!stack->IsEmpty());  
    LONGS_EQUAL(10, stack->Pop());  
    CHECK(stack->IsEmpty());  
}
```

The header file changes:

```
//added  
int values[10];
```

How many entries should there be in a stack? I think it should be decided at construction time. Instead of taking the tangent to get an allocated buffer we made a simplifying assumption and finished the test. Maybe we should do capacity next.

The cpp file changes:

```
void Stack::Push(int value)  
{  
    values[index++] = value;  
}  
  
int Stack::Pop()  
{  
    return values[--index];  
}
```

Notice there are no boundary checks. Those will go in when we have written the tests for them. By the way: I got the --index backwards at first. Those kinds of mistakes are very common, and can be very difficult to find in the presence of thousands of lines of code.

My tests yelled at me like this:

```
.
Failure in TEST(Stack, PushPopGeneralCase)
  StackTest.cpp(55)
    expected <12>
    but was <0>

..
Failure in TEST(Stack, PopReturnsWhatWasPushed)
  StackTest.cpp(40)
    expected <12>
    but was <4>

...
Errors (2 failures, 6 tests, 6 ran, 5 checks, 0 ignored, 0 filtered out)
```

After fixing that small mistake I was back with all tests running.

```
.....
OK (6 tests, 6 ran, 10 checks, 0 ignored, 0 filtered out)
```

This TDD walk through was intended to show the tight feedback loop experienced and the high test coverage that is possible when doing TDD. Stack is a very simple example, and starting with something simple is a good way to learn a new skill. Give it a try, finish this example.

---

[GANSLE] Ganssle, Jack, The Art of Designing Embedded Systems, Butterworth-Heinemann, Woburn MA, p.48

[GREN-AGL] Grenning, James, Agile Software Development for Embedded Software, ESC-349, San Jose 2007

[BECK] Beck, Kent, Test Driven Development By Example, Addison Wesley, 2003

[MACKINNON] Tim Mackinnon, Steve Freeman, Philip Craig, Endo-Testing: Unit Testing with Mock Objects (tim.mackinnon@pobox.com, steve@m3p.co.uk, philip@pobox.com)

[CPPTEST] CppTestTools are open source and can be found at <http://www.fitness.org/FitServers.CppFit.CppTestTools>

[GRENNING] Grenning, James, Progress Before Hardware, 2004  
<http://www.objectmentor.com/resources/articles//ObjectMentor/resources/articles/ProgressBeforeHardware>