

Progress before hardware

By: James Grenning

A common problem facing embedded software engineers is the concurrent development of hardware and software. The embedded software engineer does not have a test bed for their work often until late in the project. I have seen too many project plans that show an integration and test phase late in the project where hardware and software are brought together. Those integrations usually end up turning into seemingly endless debug sessions. We may tell ourselves that this project will be different, that we can integrate, test and ship in two weeks. But we'd be kidding ourselves.

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.”^[GANSLSLE] Jack goes on to say that “Test and integration are no longer individual milestones; they are the very fabric of development.”

Does the lack of the target platform mean we cannot test our code? Does that keep us from following Jack's advice and the advice from the agile development community? The answer to these questions is a resounding “No!”. In this article I'll describe how to make progress prior to hardware availability.

Embedded Software Development

Developing software is hard. Too often projects are late, with poor quality and inadequate feature sets. Embedded software development shares some of the same problems with non-embedded software development, but it also presents some additional problems. The development machine architecture and operating environment are often different from the target machine. The hardware for the target machine is usually developed concurrently with the software, and therefore not available until late in the project. The hardware may go through several iterations, changing in ways that confound the software systems. There may be real-time constraints, concurrent processing, and safety issues. Typical human-computer interfaces are not used and the computer operating the machine is hidden from the user. Resource constraints such as limited memory space or processing power are the norm.

Practices

Test driven development and object oriented design are two practices that can help make concrete progress early in the embedded software development cycle. Test driven development is an incremental technique for concurrently writing and testing code. In this article we'll look at applying TDD to embedded development.

Object Oriented Design is not a new technology, but it is a poorly understood and therefore an underused technology in the embedded development world. Object oriented languages like C++ or Java really enable this technology, but the ideas behind OOD ideas can be implemented in procedural languages such as C.

TDD and OOD can give the embedded software engineer some advantages. One specific advantage is designing, coding and testing prior to target hardware availability. I'll discuss how you can make significant progress by testing on your development machine. This implies using a portable programming language. If your environment is so constrained that you must develop in assembler you may not be able to use all the advice in this paper.

Development Environment and Execution Environment

In embedded systems the development environment usually differs from the target execution environment. I can buy a development environment at the local computer store or on the net. I can buy compilers, debuggers, source control tools, word processors and other tools for my development environment. Development environments are relatively cheap. On the other hand the target is custom made. Maybe the target is a cell phone, an engine controller, or a high speed color printer. I can't go down to the local computer store to buy that platform. Target systems are limited and expensive.

I've seen prototype hardware that cost over \$1 million. This results in the engineering team having a one to many ratio of target machines to developers. A limited resource means sharing and sharing means waiting. Waiting kills productivity. Even with access to target hardware development time is slowed whenever we test on it. Downloading and running in the target takes time, and it's a tough environment to debug in.

That said, testing in the target is necessary, but not always possible or practical. Fortunately, there are alternatives. You may be able to run in a simulator, a limited hardware prototype, or your development system.

Simulators

Simulators can be very expensive and complicated. Simulation can be done at many different levels. We can simulate the processor. We can simulate the behavior of the environment. A comprehensive simulator can rival the target platform in complexity. Later in this paper I'll describe an alternative that I call a scenario simulator.

Limited Hardware Prototype

If you cannot have the full-fledged prototype, a limited hardware prototype is very useful. The limited prototype would be very close in design to the target, but would not have all

the capabilities of the target system. Maybe it's the target processor with none of the special IO.

Using a prototype can have a very positive impact. Only part of the IO is available, so it will be necessary to build hardware independence into your design. This is one way that Object Oriented Design fits in. OOD allows the definition of interfaces, isolating one part of the system (the main application logic) from some other part of the system (the hardware implementation).

A limited prototype is a very valuable and necessary tool when the full target is not available. This prototype may suffer from the same problems as the actual target. It may be expensive, not ready, buggy, or slow for download and test. What's an engineer to do? Perhaps we can focus our testing efforts on the development system.

Development System as a Test Bed

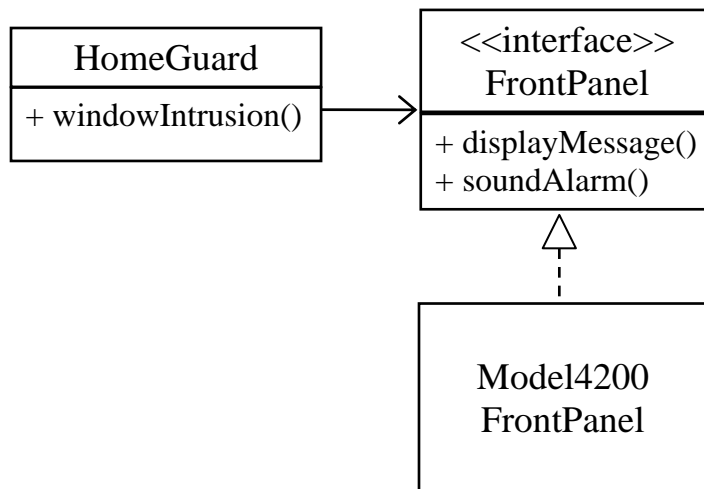
I'll start out with a claim: significant progress can be made on the development system. With isolation from hardware and operating system dependencies much of your embedded application can be tested on your development system. You'll need to be able to compile and generate executables for the development system as well as for the target.

How does this work? The development system does not have the specialized IO that the target has. How can you test it? What does running it on the development system mean?

One key to solving this problem is to design in hardware independence using OOD. The idea we started talking about a few paragraphs ago. The second key is Test Driven Design.

Object Oriented Design

When thinking about Object Oriented Design (OOD) think *interfaces*. An interface can be defined that describes how to interact with some hardware provided service. The code in the layer above the hardware isolation layer can be designed to have very limited knowledge of the underlying hardware. In C++ a class is defined that specifies the calling conventions of the interface and reveals none of the details. The main application code interacts with the execution environment through a set of interfaces. The application code can interact with the real hardware or some stand-in for the hardware that obeys the same interface.

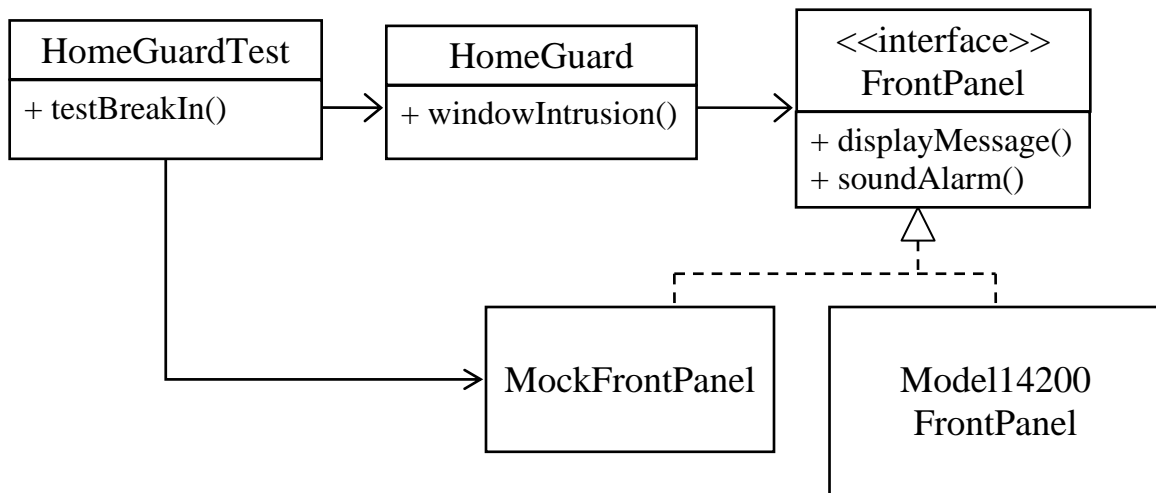


This UML diagram illustrates part of a home security system called HomeGuard. The logic in the HomeGuard class understands what it means to be a home security system. It knows the incoming events (window intrusion) and it knows how to report the security system state to its front panel. It does not know that when you write a one to address 0xFDAF00, bit 3 that the alarm will start sound. The presence of the FrontPanel interface means we can create other implementations of the FrontPanel. For instance we could create a LoggingFrontPanel that prints the changes to the FrontPanel to a log file

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. In TDD we try to test each function in isolation and incrementally build larger groups of collaborating functions and classes to provide the desired functionality. Tests come in layers. The need to test in isolation means we have to decouple one part of the system from another. Interfaces are one of our tools. Interfaces are used to decouple the parts of the system from each other.

Notice the structure of the test code and application code below. The HomeGuardTest class defines the tests (only one shown by name). HomeGuard encapsulates the security system rules. The FrontPanel describes what can be asked of a front panel. The Model4200FrontPanel implements the FrontPanel interface and knows how to interact with the hardware. But what is a MockFrontPanel? It is a test stub. It is part of the test code. When HomeGuardTest wants to test the break-in scenario, it binds HomeGuard with a MockFrontPanel. The MockFrontPanel can intercept messages meant to go to the front panel so HomeGuardTest can see if HomeGuard has responded per the requirements. The test can interrogate the Mock Object^[MACKINNON] to see what state it is in. The practice of testing helps to improve modularity. Modules are tested in isolation and in collaboration with other modules. Between the test and the Mock Object we are creating a simulator for a specific scenario.



The window intrusion test looks like this:

```

TEST(HomeGuard, WindowIntrusion)
{
    MockAlarmPanel* panel = new MockAlarmPanel();

    HomeGuard hg(panel);

    hg.arm();
    hg.windowIntrusion();
    CHECK(true == panel->isArmed());
    CHECK(true == panel->isAudibleAlarmOn());
    CHECK(true == panel->isVisualAlarmOn());
    CHECK(panel->getDisplayString() == "Window Intrusion");
}
  
```

Embedded TDD Cycle

Kent Beck, author of *Test-Drive Development by Example*^[BECK] 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. Every few minutes you find out if the code you just write is doing what you want. Is such a rapid feedback cycle feasible in embedded development? Let's look at some possibilities.

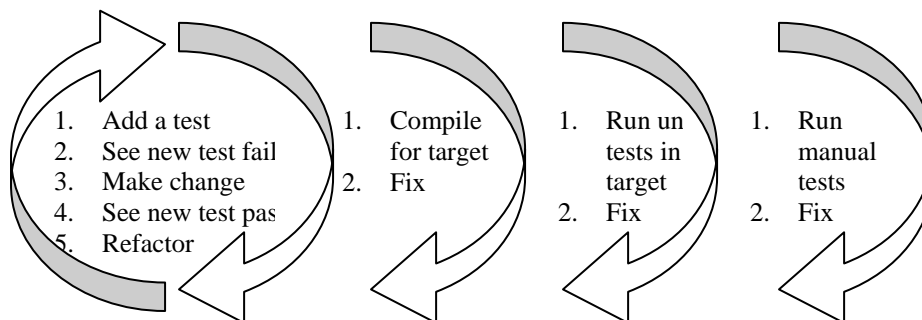
When and where are these tests run? The short answer is as often as possible and anywhere you can. Let's look at a few different situations: prior to target hardware, limited prototype hardware available and full target hardware available.

If it is early in the project cycle and we do not have target hardware, we could run our tests on our development system, with interfaces mocked out to isolate the application from the hardware. We could use the development system's native compiler. This sounds dangerous due to compiler variation; so, I would add another step to the embedded TTD cycle: periodically compile with the target's cross-compiler. This will tell us if we are marching down a porting problem path. 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 must be done at least before any check in, and probably whenever you try out some new language feature you have not compiled before.

Once we have a limited prototype, we'll continue to use the development systems as our first stop for testing and periodically compile for the target as above. We get feedback more quickly and have a friendlier debug environment. Now we'll periodically run the unit tests in the prototype. This assures that the generated code for both systems works the same. The test should be run at least prior to check-in, and more frequently based on how long it takes and how much work is being risked.

If some of the real IO is available on the limited prototype we'll start to add some tests for the hardware or that use the hardware. Automated tests are more difficult to create when the real hardware is being used. The tests may involve external instrumentation or manual verification. We want to make our tests easy to run or they will not be executed. This leads to a design where the hardware dependent code is very thin. Our goal is to automatically test most of the system.

Embedded TDD Cycle



The discussion for the full target hardware is much like the discussion for the limited prototype; except that now we can do end-to-end testing. Ideally the end-to-end testing

would be automated, but this is often difficult to achieve. One big challenge in end-to-end testing is running the system through all the scenarios it has to support. Rare scenarios have to work, but how do we get the system into a particular state and have the right triggering event to occur? Controlling the state and triggering certain events will be easier in our test environments. Our Mock Objects can be instructed to give any response needed to exercise the code. A common place to end up is that the end-to-end test is a subset of all the supported scenarios that demonstrate that the parts of the system are talking to each other properly. A combination of automated and manual tests is needed. The development systems tests never become obsolete, even though the real test bed is available.

Summary

Using Object Oriented Design Test Driven Development can provide embedded software engineers a valuable test bed for their software. These techniques can be used almost out of the box for embedded software development. But some additional steps are needed. If I have made this sound too easy, keep in mind that there are some significant challenges that have not been covered: issues of concurrency, timing constraints, testing a large application and how this fits in the bigger picture. I'll address these issues in another paper.

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

[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)

[BECK] Beck, Kent, Test Driven Development By Example, Addison Wesley, 2003, P.1