Embedded Test Driven Development Cycle

By: James Grenning

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

Test Driven Development is a technique that concurrently develops automated unit and acceptance tests and the working code that satisfies those tests. This technique was developed in the non-embedded world. Can embedded developers successfully adopt the practice for the development of embedded software? I think so and this paper outlines a variation of TDD for embedded development. TDD is usually used with Object Oriented languages, although it can be used with procedural languages such as C.

Development Environment and Execution Environment

In embedded systems 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.

I've seen prototype hardware systems costing over $1 million. This results in the engineering team having a many to one ratio of developers to target machines. This 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 difficult and expensive environment to debug in.

That said testing in the target is necessary, but not always possible or practical. Fortunately, there are alternatives.

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*[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

James W. Grenning
grenning@objectmentor.com

This cycle is designed to take only a few minutes. Every few minutes you find out if the code you just wrote is doing what you want. The tests are automated. All tests are re-run with every change. 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.

In TDD we try to test each function in isolation and incrementally build larger groups of collaborating functions and objects 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. Object oriented languages provide an advantage; interface specification is built into the language. You can also adopt conventions in C for interface definitions, although you get very little help from the language.

Embedded TDD Cycle

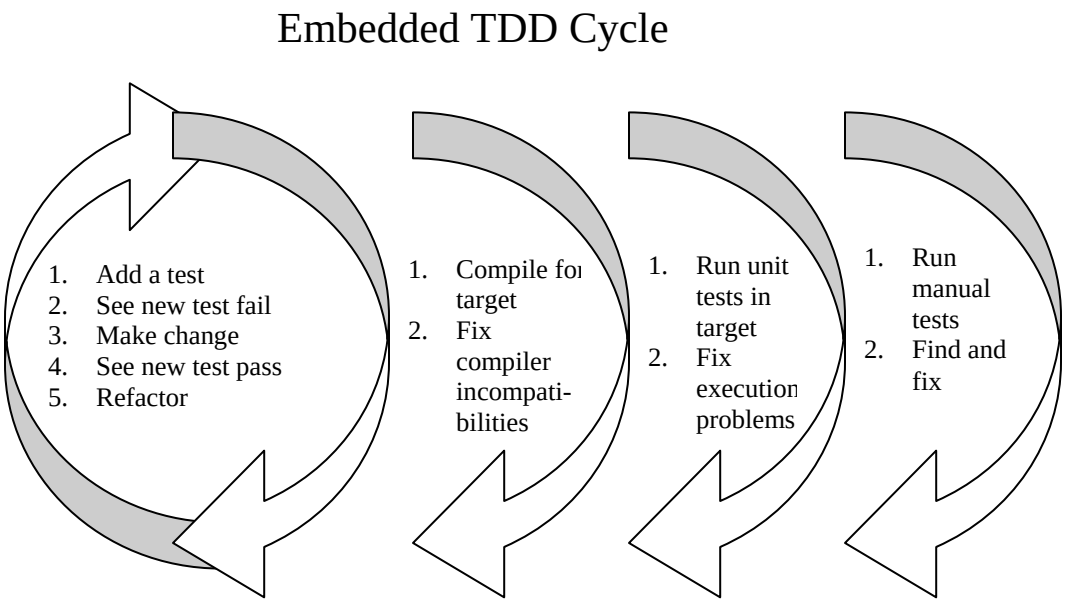
How are tests run if the target hardware is not available, or is shared between many developers? Many of the tests are designed to run on the development system. To do this the hardware dependencies have to be isolated and faked out implementations provided. That's a topic for another paper so please take a look at another paper "Progress before hardware"[GRENNING].

Using the development system's native compiler introduces a new risk: compiler differences. To mitigate this risk let's 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 path to 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.

Once we have our target hardware, we'll continue to use the development systems as our first stop for testing and periodically compile for the target as just discussed. 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 mitigate this risk we'll periodically run the unit tests in the prototype. This assures that the generated code for both systems works 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. With limited memory in your target, test may have to be run in batches.

As the target IO becomes available we'll start to add tests for the hardware or tests for code that uses 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.



Once we have a fully functional target platform 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 at exactly the right time? Controlling the system state and triggering certain events will be easier in our test environments. Our faked out hardware simulations can be instructed to give the responses 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. The end-to-end tests demonstrate that all parts of the system are talking to each other properly. A combination of automated and manual tests is needed. The development system's automated tests never become obsolete, even though the real test bed is available.

Summary

Using this Embedded Test Driven Development Cycle can provide embedded software engineers a valuable test bed for their software. Keep in mind that there are some significant challenges that have not been covered such as concurrency and timing constraints.

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

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

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

James W. Grenning
grenning@objectmentor.com