Extracted from:

Test-Driven Development for Embedded C

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

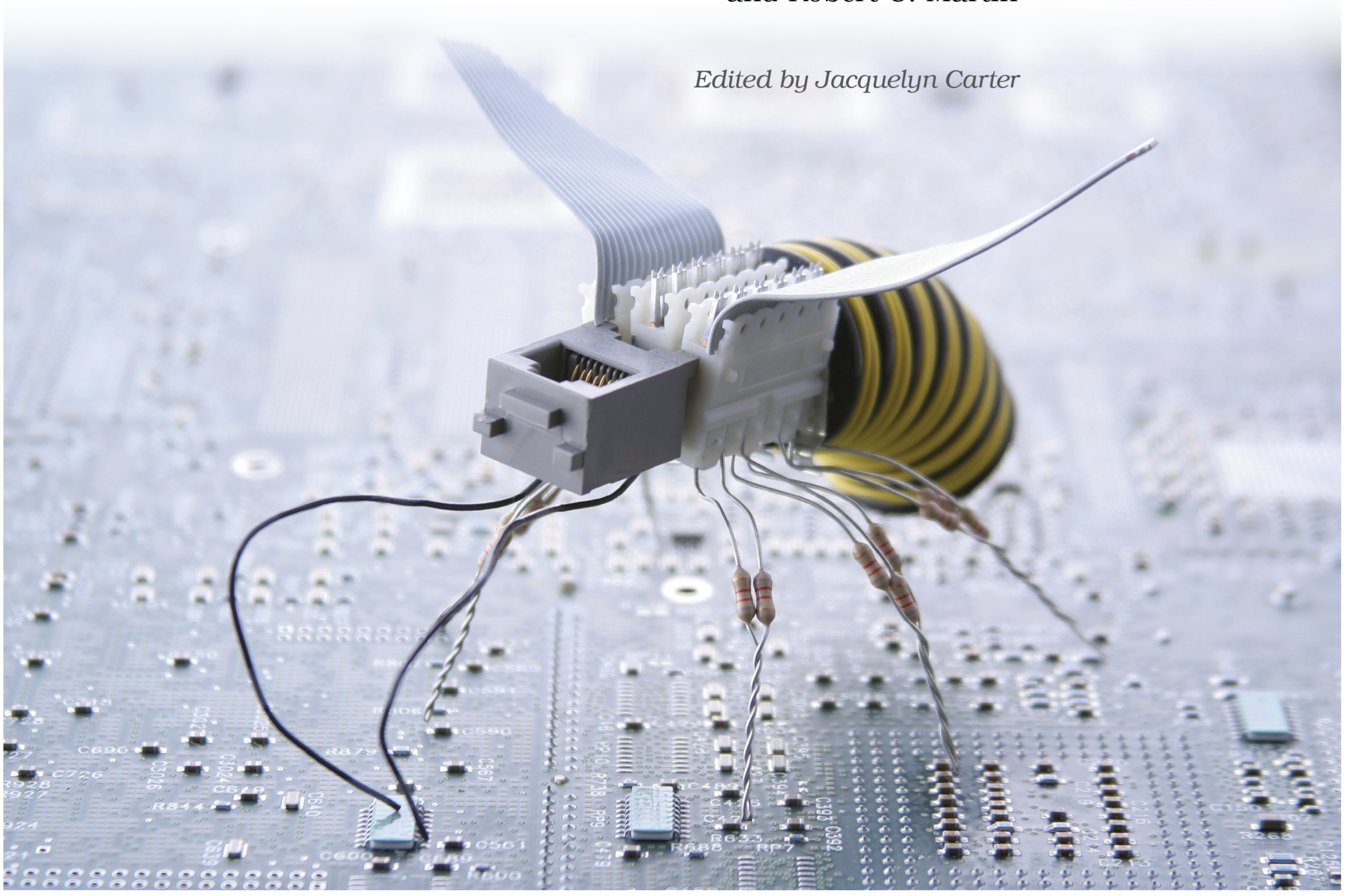
The Pragmatic Programmers

Test-Driven Development for Embedded C

James W. Grenning

Forewords by Jack Ganssle
and Robert C. Martin

Edited by Jacquelyn Carter



Test-Driven Development for Embedded C

James W. Grenning

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The previous chapter shows how an LED driver, a hardware-dependent piece of code, is developed using TDD and tested off the target on the host development system. You may wonder, are these tests valid when they are not run on target hardware? They are valuable, but along with the benefits, there are risks that must be considered and contained.

Testing off the target hardware also allows difficult-to-cause errors to be easily injected. Without this ability, a lot of code may go untested until that fateful day when the hardware error we anticipated occurs but the corrective action is wrong.

In this chapter, we'll look at specific progress blockers and time wasters common for embedded software and how to adapt TDD to help eliminate the target hardware bottleneck.[1]

5.1 The Target Hardware Bottleneck

Concurrent hardware and software development is a reality for many embedded projects. If software can be run only on the target, you will likely suffer unnecessarily from one or more of these time wasters:

- Target hardware is not ready until late in the project, delaying software testing.
- Target hardware is expensive and scarce. This makes developers wait and build up mounds of unverified work.
- When target hardware is finally available, it may have bugs of its own. The mound of untested software has bugs too. Putting them together makes for difficult debugging, long days, and plenty of finger pointing.
- Long target build times waste valuable time during the edit, compile, load, and test cycle.
- Long target upload times waste valuable time during the edit, compile, load, and test cycle.
- Long target upload times lead to batching numerous changes in one build, which means that more can go wrong, leading to more debugging.
- Compilers for the target hardware are typically considerably more expensive than native compilers. The development team may have a limited number of licenses available, adding expense and possible delays.

1. This chapter is based upon work previously published through the *Agile Times [Gre04]*, *Embedded Systems Conference [Gre07]*, and *IEEE [Gre07a]*.

Not all development efforts suffer from all those problems. But it is likely that every embedded development effort suffers from at least some of them, and these problems will block software development progress. Bob Martin's prime directive says, "We will not be blocked!"[2] Don't allow our progress to be blocked by lack of the target hardware. Don't wait for a long tool chain to do its job. Don't wait for a long upload. Don't wait in line to test your code.

Embedded developers have traditionally turned to the evaluation board for relief from one of the causes of the target hardware bottleneck.[3] An eval board provides an execution environment prior to target availability or when target hardware is too expensive for each developer to have their own. It's a very useful weapon in the embedded developer's arsenal, defending you from late and defect-laden projects, but it's really not enough. Eval boards suffer from long build and upload times but do provide a platform that works and is relatively inexpensive. Developers could have one for themselves early in the development cycle.

Here is where your development system and *dual-targeting* come into play as an effective way to cope with the target-hardware bottleneck.

5.2 Benefits of Dual-Targeting

Dual-targeting means that from day one, your code is designed to run on at least two platforms: the final target hardware and your development system. In the LED driver example, the code is ultimately intended to run on an embedded target, but first it is written and tested on the development system. The goal is not some esoteric or academic pursuit; it is a pragmatic technique to keep development going at a steady pace. We avoid the waste and risk that comes with creating an inventory of unverified work. Nancy and Ron, in the following story, describe how they put dual-targeting to work on their project.

Excerpt from: *Taming the Embedded Tiger [SM04]*

by: Nancy Van Schooenderwoert and Ron Morsicato

When you try to run newly written software on your embedded platform, you are tackling many unknowns simultaneously. A problem on the board, the CPU circuitry, the connectors, or the cabling can masquerade as a software bug, sending you off on a huge and frustrating waste of time. Hardware that worked perfectly one minute can be buggy the next—intermittent hardware bugs are horrendous to deal with. We needed a practical way to completely isolate the software under test to avoid debugging hardware and software simultaneously!

2. http://butunclebob.com/ArticleS.UncleBob.ThePrimeDirectiveOfAgileDevelopment
3. An eval board is a circuit board used in development with the same processor configuration as the target system and ideally some of the same I/O.

> Our application ran on a desktop PC as well as on the target CPU. We maintained this capability throughout development, even after we had good hardware. With so many hardware components at early stages in their own development, we simply could not risk having to troubleshoot with multiple unknowns. Very little of the application had to interact directly with hardware.
>
> This test technique required all the team members to have a clear understanding of the boundary between "pure" code and hardware-specific code. That, in itself, was good for software design and modularity. Finally, by continuing with the dual-targeting strategy, we were able to maintain an environment that was amenable to automation.

Dual-targeting solves several problems. It allows you to test code before the hardware is ready, and you can avoid the hardware bottleneck throughout the development cycle. You also avoid the finger pointing that goes with simultaneous hardware software debugging. It is a practice that keeps you moving fast.

Dual-targeting, like TDD, has another benefit: it influences your design. Paying attention to the boundaries between software and hardware produces more modular designs, in other words, designs with hardware independence. Unless you are building a one-of-a-kind product, hardware independence will remove some of the burden in future platform migrations. Hardware will change, that's a given. When it does, you'll be better prepared, having automated unit tests and code that already runs on multiple target platforms.

5.3 Risks of Dual-Target Testing

Testing code in the development system builds confidence in your code before committing it to the target, but there are risks inherent in the dual-target approach. Most of these risks are because of differences between the development and target environments. These include:

- Compilers may support different language features.
- The target compiler may have one set of bugs, while the development system native compiler has another set of bugs.[4]
- The runtime libraries may be different.
- The include filenames and features may be different.
- Primitive data types might have different sizes.
- Byte ordering and data structure alignments may be different.

Because of these risks, you may find that code that runs failure free in one environment experiences test failures in other environments.

4. The day this paragraph was written, a popular open source compiler had 3,427 open bug reports. Seventy-four new bugs arrived the previous week, while fifty-four bugs were closed. The bugs were winning by twenty.

Dual-Target Bonus Benefits

A side benefit of dual-targeting for test purposes is that the code will be easier to port in the future to different hardware platforms. How many of you are working with ten- or fifteen-year-old code that has been ported to several platforms? It's a significant issue for embedded. Hardware changes, often out of our control, will happen. Starting with dual-targeted code might just make it easier to move your code to the next unforeseen target hardware platform.

In addition, when the time comes to port your code to yet another platform, you have the tests to support the porting effort, helping to lock in the desired behavior.

The fact that there are potential differences in execution environments should not discourage you from dual-targeting. On the contrary, these are all workable obstacles on the path to getting more done. But it's best to take this path with eyes open and knowledge of some of the spear-filled pits that await further down the path.

With the benefits and risks enumerated, let's see how the embedded TDD cycle overcomes the challenges, without compromising the benefits.

5.4 The Embedded TDD Cycle

The embedded TDD cycle is an extension of the core TDD microcycle, described in Section 1.4, *The TDD Microcycle,* on page ?. It is designed to overcome the target-hardware bottleneck.

TDD is most effective when the build and test cycle takes only a handful of seconds. A longer build and test time usually results in taking bigger steps; with the bigger steps come more things that can be broken, leading to more debugging when the test finally is run. The need for a fast feedback loop leads us to move the TDD microcycle off the target to run natively on the development system. The TDD microcycle is the first stage of the embedded TDD cycle, as depicted in Figure 8, *The embedded Test-Driven Development cycle,* on page 9.

Stages 2–4 are designed to mitigate the risk of using the development platform to run unit tests. Stage 5 makes sure that the fully integrated system delivers working features. Without the TDD approach, stage 5 is where many embedded testing efforts begin.

Let's look at each stage in a little more detail.

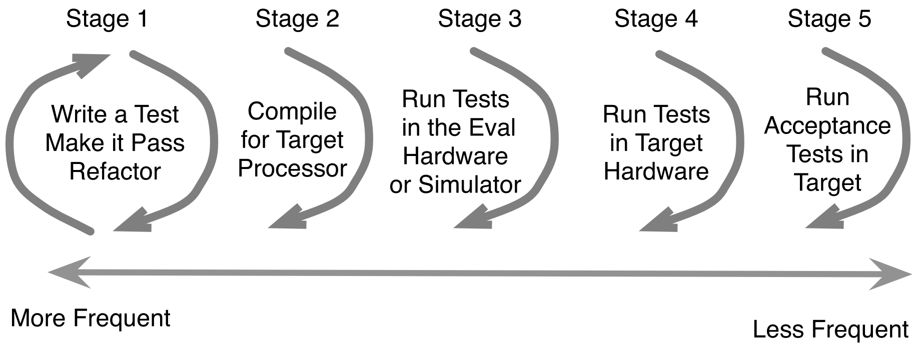


Figure 8— The embedded Test-Driven Development cycle

Stage 1—TDD Microcycle

The first stage is run most frequently, usually every few minutes. During this stage, you write the bulk of the code and compile it to run on your host development system. Testing it on the development system gives fast feedback, not encumbered by the constraints of hardware reliability or availability. There are no target compiles or lengthy uploads delaying feedback. The development system is a proven and stable execution environment; it often has a richer debugging environment (which you won't use much[5]) than the target. You will also be able to run your code through tools like `valgrind`, `profil`, and `gcov`. Also, each developer has a development system or can get one tomorrow.

During this stage, you write code that is platform independent. You look for opportunities to disconnect software from hardware, as much as is practical. The boundary between hardware and software becomes evident and is recorded in your test cases.

As mentioned earlier, there is a risk to running code on the development system when it is eventually going to run in a foreign execution environment. It's best to confront that risk regularly because sometimes there are problems...enter stage 2.

Stage 2—Compiler Compatibility Check

Periodically, compile for the target, using the cross-compiler you expect to use for production compilations. This stage is an early warning system for compiler incompatibilities. It warns of porting problems such as unavailable header files, incompatible language support, and missing

5. You won't use your debugger as much because TDD reveals mistakes as you make them. The cause is usually obvious, not requiring a debugger.

language features. This leads to code that uses only those facilities available in both development environments.

Early in an embedded development project, the tool chain may not yet be decided, and you may think this stage cannot be executed. Take your best guess about the tool chain, and compile against that compiler. You could use your suite of tests as part of your compiler evaluation criteria. As the compiler market changes, you also could use your suite of tests to evaluate new compiler vendors or versions.

You don't need to run stage 2 with every code change. You should do a target cross-compile whenever you use some new language feature, include a new header file, or make a new library call.

That said, it's best to make this happen automatically as part of a nightly build or your *continuous integration* build, where builds run on every check-in. See *Continuous Integration*, on page 12.

Stage 3—Run Unit Tests in an Eval Board

There is a risk that the compiled code will run differently in the host development system and the target processor. To mitigate this risk, run the unit tests on an eval board. Using an eval board shows when the code's behavior differs between the development system and the target processor. You will see, in a true story coming up in *Runtime Libraries Have Bugs*, on page ?, that this risk is real.

In an ideal world, we'd have the target hardware, and we would not need to use eval hardware. If it's late in the development cycle, we may have reliable target hardware making this stage appear unnecessary. So if every developer has ready access to the target hardware and we have high confidence in the hardware, this stage could be eliminated. But don't make this decision lightly.

Having the ability to run in an eval board may come in handy even after the target is ready. If there is some suspicious target behavior, you could quickly rule in or out target hardware problems by running tests in the eval platform.

These test runs should be built into the continuous integration build and run at least daily.

Stage 4—Run Unit Tests in the Target Hardware

The objectives here are the same as stage 3 while exercising the real hardware. One additional aspect to this stage is that you could also run target hardware-specific tests. These tests allow you to characterize or learn how the target hardware behaves.

An additional challenge in this stage is limited memory in the target. You might find that all the tests do not fit into the target. In that case, you can organize the tests into separate test suites where each suite fits in memory. This does result in more complicated build automation.

Stage 5—Run Acceptance Tests in the Target

Finally, we make sure the product features work by running automated and manual acceptance tests in the target. Here you have to make sure that any of the hardware-dependent code that can't be fully tested automatically is tested manually. You already know what that is.

At different points in the project life cycle, some of the stages might be either impossible or not so critical. For example, when there is no hardware early in the project, stages 4 and 5 are not possible to complete. Similarly, if the target is available and appears reliable, the eval board tests could be suspended until there is some question of target reliability. Off-target TDD, stage 1, is still where the bulk of the code is written and tested regardless of target availability.

Continuous Integration

Continuous integration is a companion practice of Test-Driven Development. In continuous integration, team members integrate and check in changes to their version control system main branch regularly, usually many times a day. As a precondition to check in, all tests must pass.

An automated build is needed for successful CI. It has to be easy to build the system. If the build is a tedious manual process with numerous mouse clicks and file copies, you won't build as often as you should. The goal is a single command build.

In the dual-target approach suggested in this chapter, the test build must be automated. But that's not the end of it. The production build should also be automated. In the mouse-heavy IDEs of today, this may take some doing.

With a single command build, you can automate the running of the build. The current state of the art is the *continuous integration server*. The CI server watches for check-ins to the code repository and initiates a complete build and test sequence once the check-in is complete. If a build breaks or any test failures occur, the team is notified usually by email. Fixing the build becomes the number-one responsibility of the team.

An embedded build would be done in two stages, first for the development system tests. If successful, the target build would run next. If your product deploys to more execution platforms, you would want a build for each.

CI is a risk reduction strategy and a time-saver. When developers go for long periods of time without integrating, the difficulty and risk of the integration grows. Like TDD, if testing is hard, do it all the time—it gets easier. With CI the mind-set is similar. If integration is hard, do it all the time. You avoid those long and error-prone code merges. Merges are smaller, and they are assisted by automated tests created via TDD.

There are good open source tools, such as CruiseControl and Hudson, to help automate CI builds and error notifications.