Fabric of Development

"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. Test and integration are no longer individual milestones; they are the very fabric of development."

—From *The Art of Designing Embedded Systems [Gan00]*, by Jack Ganssle

to be. Test-Driven Development is one way, an effective way, to weave testing into the fabric of software development. It's Kevlar for your code.[1]

There's a lot to applying TDD to embedded C, and that's what this book is about. In this chapter, you will get the 10,000-foot view of TDD. After that, you'll apply TDD to a simple C module. Of course, that will lead to questions, which we'll address in the following chapters. Before we begin, let's look at a famous bug that could have been prevented by applying TDD.

1.1 Why Do We Need TDD?

Test-Driven Development might have helped to avoid an embarrassing bug, the *Zune bug*. The Zune is the Microsoft product that competes with the iPod. On December 31, 2008, the Zune became a brick for a day. What was special about December 31, 2008? It's New Year's Eve and the last day of a leap year, the first leap year that the 30G Zune would experience.

Many people looked into the Zune bug and narrowed the problem down to this function in the clock driver. Although this is not the actual driver code, it does suffer from exactly the same bug:[2]

src/zune/RtcTime.c

```
static void SetYearAndDayOfYear(RtcTime * time)
{
    int days = time->daysSince1980;
    int year = STARTING_YEAR;
    while (days > 365)
    {
        if (IsLeapYear(year))
        {
            if (days > 366)
            {
                days -= 366;
```

1. Kevlar is a registered trademark of DuPont.
2. The actual Zune code could not be used because of copyright concerns. Zune is a registered trademark of Microsoft Corporation.

```
                year += 1;
            }
        }
        else
        {
            days -= 365;
            year += 1;
        }
    }

    time->dayOfYear = days;
    time->year = year;
}
```

Many code-reading pundits reviewed this code and came to the same wrong conclusion that I did. We focused in on the boolean expression (`days > 366`). The last day of leap year is the 366th day of the year, and that case is not handled correctly. On the last day of leap year, this code enters an infinite loop! I decided to write some tests for `SetYearAndDayOfYear()` to see whether changing boolean to (`days >= 366`) fixes the problem, as about 90 percent of the Zune bug bloggers predicted.

After getting this code into the test harness, I wrote the test case that would have saved many New Year's Eve parties:

tests/zune/RtcTimeTest.cpp

```
TEST(RtcTime, 2008_12_31_last_day_of_leap_year)
{
    int yearStart = daysSince1980ForYear(2008);
    rtcTime = RtcTime_Create(yearStart+366);
    assertDate(2008, 12, 31, Wednesday);
}
```

Just like the Zune, the test goes into an infinite loop. After killing the test process, I apply the popular fix based on reviews by thousands of programmers. To my surprise, the test fails, because `SetYearAndDayOfYear()` determines that it is January 0, 2009. New Year's Eve parties have their music but still a bug; it's now visible and easily fixable.

With that one test, the Zune bug could have been prevented. The code review by the masses got it close, but still the correct behavior eluded most reviewers. I am not knocking code reviews; they are essential. But running the code is the only way to know for sure.

You wonder, how would we know to write that one test? We could just write tests where the bugs are. The problem is we don't know where the bugs are; they can be anywhere. So, that means we have to write tests for everything,

at least everything that can break. It's mind-boggling to imagine all the tests that are needed. But don't worry. You don't need a test for every day of every year; you just need a test for every day that matters. This book is about writing those tests. This book will help you learn what tests to write, it will help you learn to write the tests, and it will help you prevent problems like the Zune bug in your product.

Finally, let's get around to answering "Why do we need TDD?" We need TDD because we're human and we make mistakes. Computer programming is a very complex activity. Among other reasons, TDD is needed to systematically get our code working as intended and to produce the automated test cases that keep the code working.

1.2 What Is Test-Driven Development?

Test-Driven Development is a technique for building software incrementally. Simply put, no production code is written without first writing a failing unit test. Tests are small. Tests are automated. Test-driving is logical. Instead of diving into the production code, leaving testing for later, the TDD practitioner expresses the desired behavior of the code in a test. The test fails. Only then do they write the code, making the test pass.

Test automation is key to TDD. Each step of the way, new automated unit tests are written, followed immediately by code satisfying those tests. As the production code grows, so does a suite of unit tests, which is an asset as valuable as the production code itself. With every code change, the test suite runs, checking the new code's function but also checking all existing code for compatibility with the latest change.

Software is fragile. Just about any change can have unintended consequences. When tests are manual, we can't afford to run all the tests that are needed to catch unintended consequences. The cost of retest is too high, so we rerun the manual tests we think are needed. Sometimes we're not too lucky and defects are created and go undetected. In TDD, the tests help detect the unintended consequences, so when changes are made, prior behavior is not compromised.

Test-Driven Development is not a testing technique, although you do write a lot of valuable automated tests. It is a way to solve programming problems. It helps software developers make good design decisions. Tests provide a clear warning when the solution takes a wrong path or breaks some forgotten constraint. Tests capture the production code's desired behavior.

TDD is fun! It's like a game where you navigate a maze of technical decisions that lead to highly robust software while avoiding the quagmire of long debug sessions. With each test there is a renewed sense of accomplishment and clear progress toward the goal. Automated tests record assumptions, capture decisions, and free the mind to focus on the next challenge.

1.3 Physics of TDD

To see how Test-Driven Development is different, let's compare it to the traditional way of programming, something I call *Debug-Later Programming*. In DLP, code is designed and written; when the code is "done," it is tested. Interestingly, that definition of *done* fails to include about half the software development effort.

It's natural to make mistakes during design and coding—we're only human. Therein lies the problem with Debug-Later Programming; the feedback revealing those mistakes may take days, weeks, or months to get back to you, the developer. The feedback is too late to help you learn from your mistakes. It won't help you avoid the mistake the next time.

With the late feedback, other changes may be piled on broken code so that there is often no clear root cause. Some code might depend on the buggy behavior. With no clear cause and effect, your only recourse is a bug hunt. This inherently unpredictable activity can destroy the most carefully crafted plans. Sure, you can plan time for bug fixing, but do you ever plan enough? You can't estimate reliably because of unknowable unknowns.

Looking at Figure 1, *Physics of Debug-Later Programming*,on page 6, when the time to discover a bug (T_d) increases, the time to find a defect's root cause (T_{find}) also increases, often dramatically. For some bugs, the time to fix the bug (T_{fix}) is often not impacted by T_d. But if the mistake is compounded by other code building on top of a wrong assumption, T_{fix} may increase dramatically as well. Some bugs lay undetected or unfound for years.

Now take a look at Figure 2, *Physics of Test-Driven Development*,on page 7. When the time to discover a bug (T_d) approaches zero, the time to find the bug (T_{find}) also approaches zero. A code problem, just introduced, is often obvious. When it is not obvious, the developer can get back to a working system by simply undoing the last change. $T_{find} + T_{fix}$ is as low as it can get, given that things can only get worse as time clouds the programmer's memory and as more code depends on the earlier mistake.

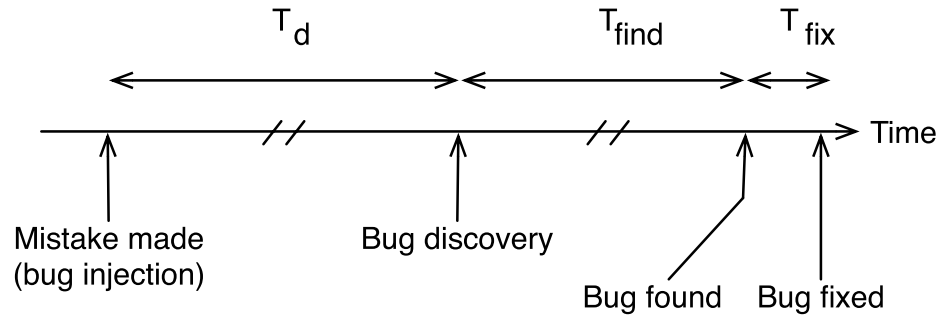


Figure 1— Physics of Debug-Later Programming

In comparison, TDD provides feedback immediately! Immediate notification of mistakes prevents bugs. If a bug lives for less than a few minutes, is it really a bug? No, it's a prevented bug. TDD is defect prevention. DLP institutionalizes waste.

1.4 The TDD Microcycle

I'll start by telling you what TDD is not. It is not spending an hour, a day, or a week writing masses of test code, followed by writing reams of production code.

TDD is writing one small test, followed by writing just enough production code to make that one test pass, while breaking no existing test. TDD makes you decide what you want before you build it. It provides feedback that everything is working to your current expectations.

At the core of TDD is a repeating cycle of small steps known as the TDD microcycle. Each pass through the cycle provides feedback answering the question, does the new and old code behave as expected? The feedback feels good. Progress is concrete. Progress is measurable. Mistakes are obvious.

The steps of the TDD cycle in the following list are based on Kent Beck's description in his book *Test-Driven Development [Bec02]*:

1. Add a small test.
2. Run all the tests and see the new one fail, maybe not even compile.
3. Make the small changes needed to pass the test.
4. Run all the tests and see the new one pass.
5. Refactor to remove duplication and improve expressiveness.

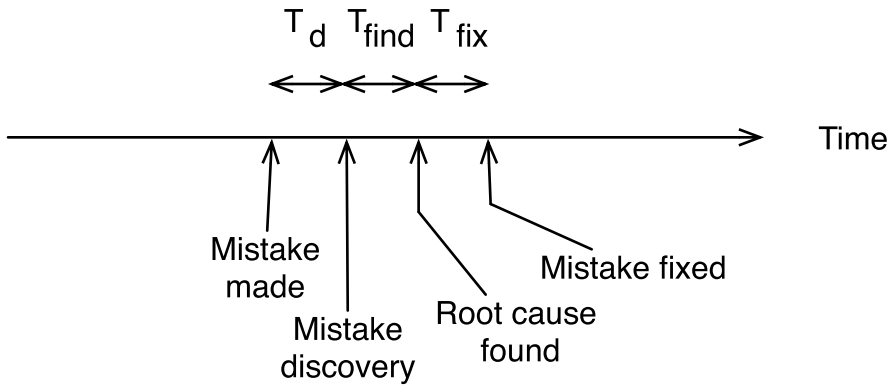


Figure 2— Physics of Test-Driven Development

Each spin through the TDD cycle is designed to take a few seconds up to a few minutes. New tests and code are added incrementally with immediate feedback showing that the code just written actually does what it is supposed to do. You grow the code envisioned in your mind from simple roots to its full and more complex behavior.

As you progress, not only do you learn the solution, but you also build up your knowledge of the problem being solved. Tests form a concrete statement of detailed requirements. As you work incrementally, the test and production code capture the problem definition and its solution. Knowledge is captured in a nonvolatile form.

With every change, run the tests. The tests show you when the new code works; they also warn when a change has unintended consequences. In a sense, the code screams when you break it!

When a test passes, it feels good; it is concrete progress. Sometimes it's a cause for celebration! Sometimes a little, sometimes a lot.

Keep Code Clean and Expressive

Passing tests show correct behavior. The code has to work. But there's more to software than correct behavior. Code has to be kept clean and well structured, showing professional pride in workmanship and an investment in future ease of modification. Cleaning up code has a name, and it's the last step of the repeating microcycle. It's called *refactoring*. In Martin Fowler's book *Refactoring: Improving the Design of Existing Code [FBBO99]*, he describes refactoring like this: refactoring is the activity of changing a program's

structure without changing its behavior. The purpose is to make less work by creating code that is easy to understand, easy to evolve, and easy to maintain by others and ourselves.

Small messes are easy to create. Unfortunately, they are also easy to ignore. The mess will never be easier to clean up than right after—ahem—*you* make it. Clean the mess while it's fresh. "All tests passing" gives an opportunity to refactor. Refactoring is discussed and demonstrated throughout this book, and we'll focus on it in Chapter 12, *Refactoring*,on page 219.

TDD helps get code working in the first place, but the bigger payoff is in the future, where it supports future developers in understanding the code and keeping it working. Code can be (almost) fearlessly changed.

Test code and TDD are first about supporting the writer of the code, getting the code to behave. Looking further out, it's really about the reader, because the tests describe what we are building and then communicate it to the reader.

You will hear TDD practitioners call the rhythm embodied by the microcycle *Red-Green-Refactor*. To learn why, see *Red-Green-Refactor and Pavlov's Programmer*,on page 9.

1.5 TDD Benefits

Just as with any skill, such as playing pool or skiing black diamonds,[3] TDD skills take time to develop. Many developers have adopted it and would not go back to Debug-Later Programming. Here are some of the benefits TDD practitioners report:

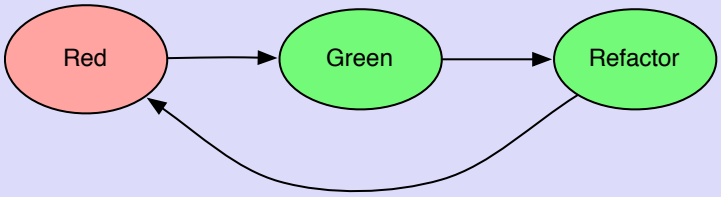
Fewer bugs
: Small and large logic errors, which can have grave consequences in the field, are found quickly during TDD. Defects are prevented.

Less debug time
: Having fewer bugs means less debug time. That's only logical, Mr. Spock.

Fewer side effect defects
: Tests capture assumptions, constraints, and illustrate representative usage. When new code violates a constraint or assumption, the tests holler.

3. Black diamond ski runs are really steep.

Red-Green-Refactor and Pavlov's Programmer



The rhythm of TDD is referred to as *Red-Green-Refactor*. Red-Green-Refactor comes from the Java world, where TDD practitioners use a unit test harness called JUnit that provides a graphical test result representation as a progress bar. A failing unit test turns the test progress bar red. The green bar is JUnit's way of saying all tests passing. Initially, new tests fail, resulting in an expected red bar and a feeling of being in control. Getting the new test to pass, without breaking any other test, results in a green bar. When expected, the green bar leaves you feeling good. When green happens and you expected red, something is wrong—maybe your test case or maybe just your expectation.

With all tests passing, it is safe to refactor. An unexpected red bar during refactoring means behavior was not preserved, a mistake was detected, or a bug was prevented. Green is only a few undo operations away and a safe place to try to refactor from again.

Documentation that does not lie
: Well-structured tests become a form of executable and unambiguous documentation. A working example is worth 1,000 words.

Peace of mind
: Having thoroughly tested code with a comprehensive regression test suite gives confidence. TDD developers report better sleep patterns and fewer interrupted weekends.

Improved design
: A good design is a testable design. Long functions, tight coupling, and complex conditionals all lead to more complex and less testable code. The developer gets an early warning of design problems if tests cannot be written for the envisioned code change. TDD is a code-rot radar.

Progress monitor
: The tests keep track of exactly what is working and how much work is done. It gives you another thing to estimate and a good definition of *done*.

Fun and rewarding
: TDD is instant gratification for developers. Every time you code, you get something done, and you know it works.

1.6 Benefits for Embedded

Embedded software has all the challenges of "regular" software, such as poor quality and unreliable schedules, but adds challenges of its own. But this doesn't mean that TDD can't work for embedded.

The problem most cited by embedded developers is that embedded code depends on the hardware. Dependencies are a huge problem for nonembedded code too. Thankfully, there are solutions for managing dependencies. In principle, there is no difference between a dependency on a hardware device and one on a database.

There are challenges that embedded developers face, and we'll explore how to use TDD to your advantage. The embedded developer can expect the same benefits described in the previous section that nonembedded developers enjoy, plus a few bonus benefits specific to embedded:

- Reduce risk by verifying production code, independent of hardware, before hardware is ready or when hardware is expensive and scarce.
- Reduce the number of long target compile, link, and upload cycles that are executed by removing bugs on the development system.
- Reduce debug time on the target hardware where problems are more difficult to find and fix.
- Isolate hardware/software interaction issues by modeling hardware interactions in the tests.
- Improve software design through the decoupling of modules from each other and the hardware. Testable code is, by necessity, modular.

The next part of the book is dedicated to getting you started with TDD. After a TDD programming example in the next couple chapters, we'll talk more about some of the additional techniques needed for doing TDD for embedded software in Chapter 5, *Embedded TDD Strategy*,on page 77.