TEST-DRIVEN DEVELOPMENT FOR EMBEDDED C: WHY DEBUG?

TEST-DRIVEN DEVELOPMENT HELPS YOU IMPROVE SCHEDULE PREDICTABILITY AND PRODUCT QUALITY BY ELIMINATING BUGS BEFORE THEY MAKE THE BUG LIST.

BY JAMES W GRENNING • RENAISSANCE SOFTWARE CONSULTING

You all write code and then toil to make it work. You build it, and then you fix it. Testing is an afterthought—something you do after you write the code. You spend about half your time in the unpredictable activity of debugging. Debugging shows up on your schedule under the guise of test and integration. It is a source of risk and uncertainty. Fixing one bug might lead to another and, sometimes, to a cascade of other bugs.

IMAGE: SHUTTERSTOCK

You keep statistics to help predict how much time you need to remove the bugs. You measure and manage the bugs. You watch for the "knee" of the curve, the trend that shows that you finally are fixing more bugs than you are introducing. The knee shows that you are almost done, but you never know whether another killer bug is hiding in a dark corner of the code.

One aspect of designing for manufacturability is determining why these bugs happen to you. The simple answer is this: You put them there. It's the way you work. When test follows development, it will find defects (**Figure 1** and **Reference 1**). You make mistakes when you develop; the tests' job is to find the defects. If you are any good at testing, you'll find bugs. Following development by test means you must find, fix, and manage a boatload of defects.

This procedure, debug-later programming, is currently the most popular way to program. Write the code; debug it later. Debug-later programming is risky. You make mistakes because you are human. You can be sure of neither when the bugs will appear nor how long it will take to find them (**Figure 2**).

When the time to discover a bug (T_D) increases, the time to find the bug's root cause (T_{FIND}) also increases—often dramatically. If it's a few hours, days, weeks, or months from introduction to discovery, you lose context and must start the bug hunt. When you find defects outside the development phase, then you must also manage the bug. For some bugs, the time to discover a bug does not affect the time to fix the bug (T_{FIX}), and some working code may also depend on the bug. Fixing such bugs in turn causes other bugs.

Short cycles and aggressive test automation save time and effort. You need not repeat tedious and error-prone manual tests. With test automation, the cost of retest can involve almost no additional effort. Test automation quickly detects side effects and avoids the need for debugging sessions.

In another approach, TDD (test-driven development), you develop test and production code concurrently in a tight feedback loop (**references 2** and **3**). In a TDD microcycle, you write a test, watch it not compile, fail to make it compile, make it pass, clean up any mess, and repeat the process until you are finished. Writing test code and writing production code are integrated processes. If you make a mistake and the new test does not pass, you immediately know about and can fix the mistake. The tests tell you whether you get the new test to pass but introduce a bug. You plug automated tests into a unit test harness (**Figure 3**). Running a retest is free.

Some but not all occurrences of bugs are prevented when you perform development and test in the TDD-feedback loop. TDD has a profound effect on design and how you spend your time.

In contrast to debug-later programming, the physics of TDD do not include the risk and uncertainty of tracking down bugs (**Figure 4**). When the time to discover a mistake approaches zero, the time to find the mistake also approaches zero. A code problem that you recently introduced is often obvious. When it is not obvious, the developer can get back to a working system by simply undoing the last change. The time for finding and fixing the mistake is as low as it can get, given that things can get only worse as time clouds the programmer's memory and as more code depends on the earlier mistake.

TDD provides immediate notification of mistakes that allow you to prevent many of the bugs you would otherwise have to track down. TDD

AT A GLANCE

- Why do these bugs happen to you? You put them there.
- In TDD (test-driven development), you develop test and production code concurrently in a tight feedback loop.
- TDD might have helped to avoid the embarrassing Zune bug.
- Target-hardware bottleneck comes in various forms, and you can use TDD to avoid the bottleneck during the tight TDD-feedback loop.
- TDD helps you ensure that your code does what you think it does. How can you build a reliable system if it does not?
- TDD quickly finds small and large logic errors, preventing bugs and ultimately yielding fewer bugs.

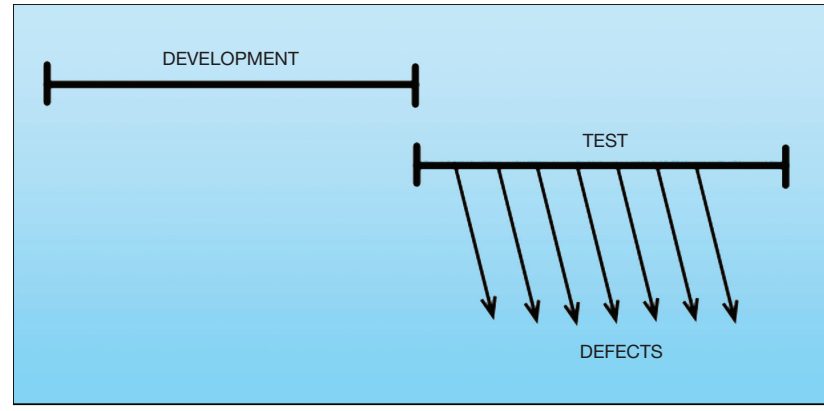

Figure 1 When test follows development, it will find defects.

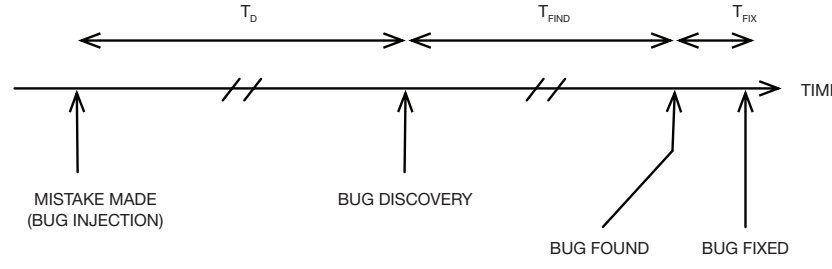

Figure 2 You make mistakes because you are human. You can be sure of neither when the bugs will appear nor how long it will take to find them.

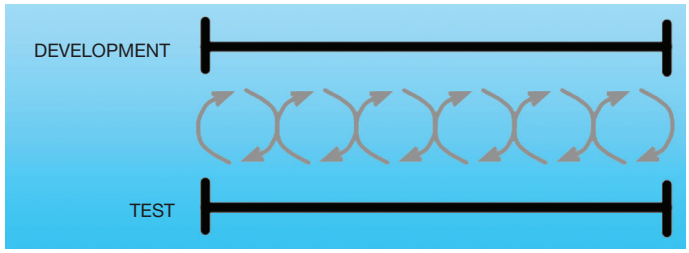

Figure 3 The tests tell you whether you get the new test to pass but introduce a bug. You plug automated tests into a unit-test harness.

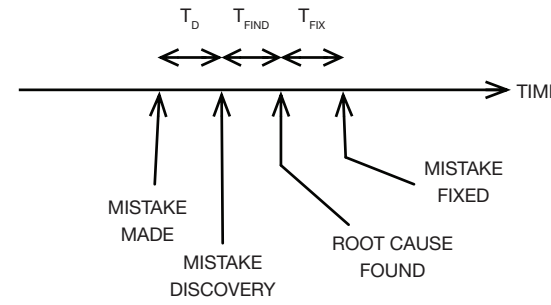

Figure 4 TDD has a profound effect on design and how you spend your time. In contrast to debug-later programming, the physics of TDD do not include the risk and uncertainty of tracking down bugs.

represents defect prevention, whereas debug-later programming institutionalizes the wasteful activity of debugging.

THE ZUNE BUG

TDD might have helped to avoid the embarrassing Zune bug. Microsoft's Zune competes with Apple's iPod. On Dec 31, 2008, the Zune became "brick for a day." Dec 31 was New Year's Eve and the last day of a leap year, the first leap year that the 30G Zune would experience. Many people narrowed down the Zune bug to a function in the clock driver. Although the code in **Listing 1** is not the actual driver code, it suffers from the same defect. Can you find the cure for the Zune's infinite loop in **Listing 1**?

Many code-reading pundits reviewed this code and came to the same wrong conclusion that you might. The last day of leap year is the 366th day of the year, and the Zune handled that case incorrectly. On that day, this function never returns! I wrote code to set the year and the day of the year to see whether changing the boolean code to days being equal to or greater than 366 fixes the problem, as about 90% of the Zune bug bloggers predicted. After getting this code into the test harness, I wrote the test case (**Listing 2**). Just as the Zune does, the test goes into an infinite loop. I applied the popular fix employing reviews by thousands of programmers. Much to my surprise, the test fails; the set-year-and-time-of-day test determines the date as Jan 0, 2009. New Year's Eve parties would still have their music, but the Zune would still have a bug.

LISTING 1 ZUNE CODE

```
static void SetYearAndDayOfYear(RtcTime * time)
{
    int days = time->daysSince1980;
    int year = STARTING_YEAR;
    while (days > 365)
    {
        if (IsLeapYear(year))
        {
            if (days > 366)
            {
                days -= 366;
                year += 1;
            }
        }
        else
        {
            days -= 365;
            year += 1;
        }
    }

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

LISTING 2 TEST CODE

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

That one test could have prevented the Zune bug. How would you know enough to write that one test? You would know if you could write tests for only where the bugs are. The problem is that you don't know where the bugs are; they can be anywhere. So that means you must write tests for everything—at least everything that can break. It's mind-boggling to imagine all the tests that are necessary. Don't worry, though; you don't need a test for every day of every year. You need a test only for those days that matter.

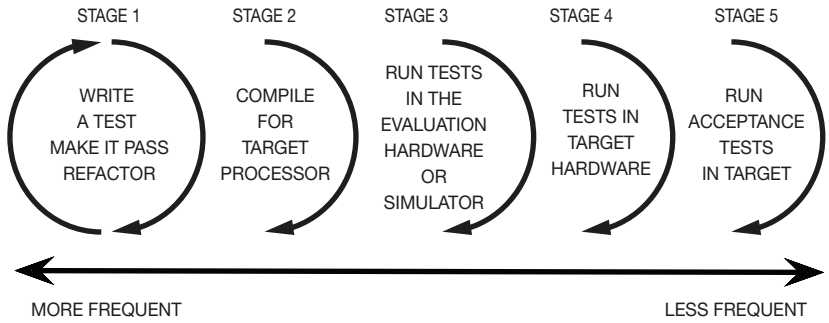

Figure 5 The need for fast feedback leads you to move the TDD microcycle off the target to run natively on the development system. A TDD cycle contains dual-target risks and provides the benefit of a fast TDD feedback loop.

Computer programming is complex. TDD systematically gets your code working as you intend and produces the automated test cases that keep the code working.

EMBEDDED DESIGN

When I first explored TDD, I realized that it could help with one of the problems—target-hardware bottleneck—that plague many embedded-software developers. This bottleneck comes in various forms, and you can use TDD to avoid the bottleneck during the tight TDD feedback loop. Concurrent hardware and software development is a reality for many embedded-development efforts. If software can be run only on the target hardware, you will likely suffer unnecessarily from at least one time waster. For example, the target hardware may not be ready until late in the delivery cycle, delaying software testing; it may be expensive or scarce; or it may have its own bugs. The target hardware may also have long development times or long uploading times. Most embedded-development teams experience some of these problems, which slow progress and reduce feedback for building today's complex systems.

To avoid the target-hardware bottleneck, you can use "dual-targeting"—designing your production code and tests so that many of them run on a standard PC. Dual-targeting has its own risks, however. Testing code in the development system builds confidence in your code before committing it to the target. Most of the risks of dual-targeting are due to differences between the development and the target environments. These differences include varying amounts of support for language features, different compiler bugs, run-time-library variations, file-name differences, and different word sizes. Because of these risks, you may find that code that runs failure-free in one environment experiences test failures in other environments.

Potential differences in execution environments should not discourage you from dual-targeting, however. On the contrary, you can work around these obstacles on the path to achieving your goals. The embedded-TDD cycle overcomes the challenges without compromising the benefits.

DEVELOPMENT CYCLE

TDD is most effective when the build-and-test cycle takes only a handful of seconds. This approach rules out having target hardware in the loop for most programmers. The need for fast feedback leads you to move the TDD microcycle off the target to run natively on the development system. **Figure 5** shows a TDD cycle that contains the dual-target risks and provides the benefit of a fast TDD feedback loop.

By going through the stages listed in **Table 1**, you expect to find problems at the appropriate stage. For example, you would expect each stage to help find these problems. Stage 1 gives you fast feedback when you are programming, ensuring that the code does what you think it is doing. Stage 2 ensures that your code compiles in both environments. Stage 3 ensures that the code runs the same in both the host and the target processor. The evaluation hardware may need a larger memory than the target does, so that the test and production code can fit into the address space. You can sometimes omit Stage 3 if you have a reliable target with the space to run the unit tests. Stage 4 runs the tests in the target. You could introduce some hardware-dependent unit tests in Stage 4. Stage 5 encompasses seeing whether your system works as it should when it is fully integrated. It's a good idea to automate at least some of Stage 5.

POTENTIAL DIFFERENCES IN EXECUTION ENVIRONMENTS SHOULD NOT DISCOURAGE YOU FROM DUAL-TARGETING. YOU CAN WORK AROUND THESE OBSTACLES ON THE PATH TO ACHIEVING YOUR GOALS. THE EMBEDDED-TDD CYCLE OVERCOMES THE CHALLENGES.

The embedded TDD cycle doesn't prevent all problems, although it should help to find most problems soon after their introduction and in an appropriate stage. You should be able to manually execute stages 2 through 4 upon check-in or at least nightly. A continuous integration server, such as Cruise Control or Jenkins, can watch your source repository and initiate builds after check-in.

TDD helps you ensure that your code does what you think it does. How can you build a reliable system if it does not? It helps you get the code right in the first place, and it creates a regression-test suite that helps you keep your code working. You waste considerable effort in finding, chasing, and fixing bugs. Many developers are now preventing these bugs from occurring with

TABLE 1 LIKELY PROBLEMS

Stage	Problems
1	Logic, design, modularity, interface, and boundary conditions
2	Compiler compatibility, including language features, and library compatibility, including header files and prototypes
3	Processor-execution problems, such as bugs in compiler and standard libraries, and portability problems, such as word size, alignment, and endian
4	Processor-execution problems, such as bugs in compiler and standard libraries, and portability problems, such as word size, alignment, and endian; hardware-integration problems; and misunderstood hardware specifications
5	Processor-execution problems, such as bugs in compiler and standard libraries, and portability problems, such as word size, alignment, and endian; hardware-integration problems; and misunderstood hardware and feature specifications

TDD. It fundamentally changes how you program.

TDD quickly finds small and large logic errors, preventing bugs and ultimately yielding fewer bugs. Fewer bugs in turn mean less debugging time and fewer side-effect defects. When new code violates a constraint or an assumption, the tests let you know. Well-structured tests then become a form of executable documentation.

TDD also gives you peace of mind because thoroughly tested code with a comprehensive regression-test suite gives confidence. Developers using TDD report fewer interrupted weekends and better sleep patterns. TDD also monitors progress, keeping track of what is working and how much work is taking place. When code changes become difficult to test, it provides an early warning of design problems. EDN

REFERENCES

1 Beck, Kent; and Cynthia Andres, *Extreme Programming Explained: Embrace Change*, Second Edition, Pearson Education Inc, 2005, ISBN: 0-321-27865-8.

2 Grenning, James W, *Test-Driven Development for Embedded C*, The Pragmatic Bookshelf, 2011, ISBN: 978-1-93435-662-3, http://bit.ly/wXvDFa.

3 Grenning, James W, "Test-Driven Development for Embedded C, Why Debug?" Embedded Systems Conference, September 2011, http://bit.ly/wi7QEX.

AUTHOR'S BIOGRAPHY

James Grenning is founder of Renaissance Software Consulting, where he trains, coaches, and consults worldwide. With more than 30 years of software-development experience, both technical and managerial, Grenning brings a wealth of knowledge, skill, and creativity to software-development teams and their management. His professional roots are in embedded software, so he is leading the way to introduce Agile development practices to that challenging world. He is the author of Test Driven Development for Embedded C *and the inventor of Planning Poker, an estimating technique, and participated in the creation of the Manifesto for Agile Software Development.*