Taming the Embedded Tiger – Agile Test Techniques for Embedded Software

Nancy Van Schooenderwoert, Ron Morsicato
Agile Rules, Lexington, MA
nancyv@agilerules.com, ronm@agilerules.com

Abstract

Strong unit testing is the foundation of agile software development but embedded systems present special problems. Test of embedded software is bound up with test of hardware, crossing professional and organizational boundaries. Even with evolving hardware in the picture, agile methods work well provided you use multiple test strategies. This has powerful implications for improving the quality of high-reliability systems, which commonly have embedded software at their heart.

Keywords. *Agile methodologies, Extreme Programming, Embedded Development, Embedded Agile, High Reliability Software, Real-time Software, Agile Testing.*

1. Introduction

This paper focuses on the testing techniques the authors used on a recent embedded real-time project. Our team and management went through a transition to Extreme Programming and experienced many of the same insights and difficulties described by other teams, but this paper will concentrate primarily on the test techniques we used. The reasons for that choice are

- Extreme Programming is almost unheard-of in embedded work, possibly because there is little guidance on how to do agile testing there
- We assembled a set of powerful test techniques and a good description of them is enough material for one paper
- Our goal with this paper is to help other real-time/embedded teams get on board with agile techniques – strong testing is where it starts!

We offer this technical paper with one caveat: We, and many other XP practitioners, have come to understand that agile software development is first a people-centered activity and only secondarily a technical discipline. XP cannot be imposed on developers who don't buy into its philosophy. It is a perfect example of what W. Edwards Deming [1] called an "intrinsic motivator" in that it depends on people's pride in their work, wanting to be part of a team, and willingness to pitch in. The opposite is what he called "extrinsic motivators" like arbitrary deadlines, threats, and bureaucratic rules. Those things can never generate real teamwork – they destroy it. You need real teamwork to get the most from the test techniques described in this paper.

2. Our Situation

The authors worked together on a recent real-time embedded project, Nancy as team technical leader and Ron as a senior member of the team.

The project's goal was to produce a mobile spectrometer, using a newly developed proprietary technology. It was a new product, being designed from scratch. The system would receive a spectrum of data, apply a complex mathematical algorithm, and send results out over a communication link to a separately supplied display unit.

Our design called for a 32-bit microprocessor with floating point. We used C with some assembler, an RTOS, and had to support two different communications protocols. The final size of the embedded code was just under 30,000 lines, excluding comments.

This project is typical of many real-time embedded projects – a data acquisition stage, application of an algorithm, followed by output of a result.

These test techniques themselves are not new, but we used them to an extent we have not seen before, and in a combination we have not seen before.

The following sections on each technique will present the problem it addresses and then discuss how well that technique worked for us. The concluding section will present our reflections on the use of these practices in combination.

2.1. Evolution of Hardware Platform

In embedded development, the hardware is always changing. It evolves in steps that the software must support. This dovetails nicely with iterative software development. Our project was using a new microprocessor, which could only be obtained as part of an evaluation board from the chip vendor. This was because the chip was not yet fully working – the CPU worked but not all of the on-board peripherals.

Figure 1 gives an idea of the hardware evolution we experienced, which is not at all unusual in embedded work. Early on, an evaluation board (often several of them from different vendors) is obtained so engineers can select a CPU and check out its performance. The figure shows configuration 1 as having a rev-A microprocessor. Later configurations add a prototype board. In our case this board was an early version of the sensor hardware for bringing in the spectrum that would be measured. The final configuration would dispense with the evaluation board and combine everything on one board for economical production costs. Ideally one would go straight from Configuration 4 to Configuration 7 if possible. It's not uncommon that the board intended to be for final production becomes instead one more pre-production board, as problems or new requirements are discovered.

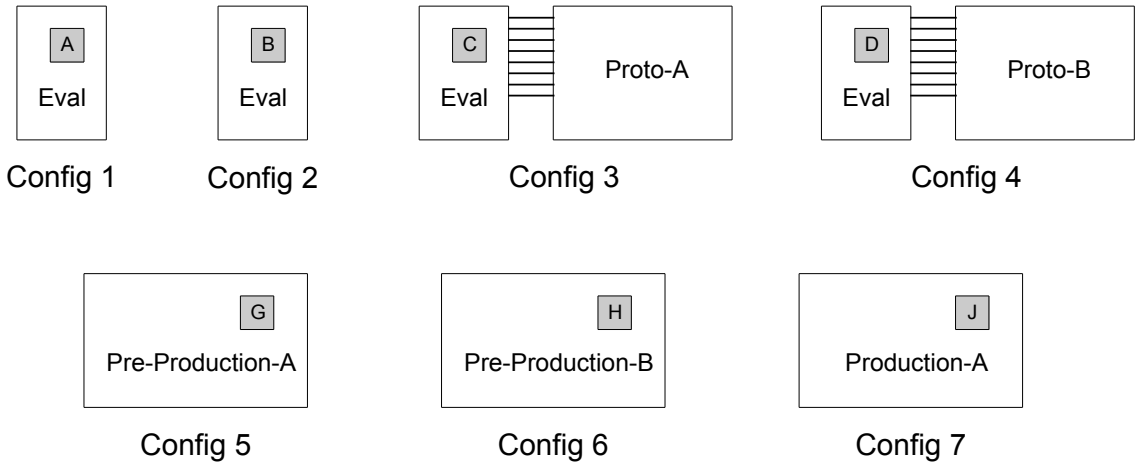

Figure 1. Evolution of the hardware

Any embedded software development strategy must deal with changing hardware. A common approach (and the one we used) is to let #define statements control parts of the code that are specific to a given hardware configuration. This strategy was used in parallel with the remaining test practices described here.

3. Embedded Testing Techniques

3.1. Trouble Log That's Always On

An ancient defense against bugs is to code a mechanism for displaying brief messages to indicate that some portion of the code was reached, or a variable has an invalid value, etc. These methods are awkward, and when enabled they can make the code execute differently,

frustrating attempts to troubleshoot a problem. The logging is normally disabled because it consumes too many system resources. Having seen things like this on past projects, we wanted to improve on it.

Early in our project, Nancy established a different kind of trouble log to avoid those disadvantages. It's an extension of the idea put forth by Steve Maguire [2] that makes use of the C library "assert" macro. The ordinary assert macro can be used to state an assumption, e.g.

```
assert( bufcount > 0);
```

so that execution will halt if the expression in the parentheses is not true. Such a halt lets you examine the call stack to see what went wrong. That's ok in the development lab, but the code shouldn't halt when it's in customer use! A wrapper function provided the smarts to either use the plain assert macro or quietly write a diagnostic message to a buffer, depending on a flag that indicated development or field environment.

The overhead of a 'printf' call is high for an embedded system, and there is typically no display device anyway. So we used a circular buffer in RAM to hold brief fixed-length text messages. Each message reported the source file, line number, severity, and diagnostic text. Here is an example of a trouble log message being written:

```
    Log( I, __FILE__, __LINE__, "Beginning
Algo run.\n" );
```

and how the resulting log file looks (below)

```
Rec #   Sev   Filename    Line #            Description              Timer Ticks
  4     I     meas_prd      654      Beginning Algo run.                17384
  5     W     stat_tbl     2536      2147565568:BIT check failed        17383
  6     I     can_comm     2421                                         17112
  7     I     meas_prd      654      Beginning Algo run.                16939
  8     W     stat_tbl     2536      2147565568:BIT check failed        16938
  9     E     meas_prd     1029      Bailing out -'prod_name' error     14951
 10     W     det_cons      601      No Product selected.               14951
```

Figure 2. Sample trouble log output

Our embedded system had a communication link to a PC that allowed upload of the trouble log. Because this logging system was the first code written, the team used it throughout all of the code to validate assumptions, and to trace execution of code that is difficult to trace any other way. For example, a diagnostic could be written at the start and finish of an ISR, and since the trouble log contained the clock tick for each entry, you could just subtract to see how long the ISR took to execute.

We'd code a severity level into each trouble log call. That was necessary to govern the behavior of the system, as follows:

Severity	Action in Development	Action in Field
Informational	Write to buffer	Write to buffer
Warning	Halt	Write to buffer
Error	Halt	Write to buffer
Fatal	Halt	Write to buffer and do orderly shutdown

Table 1. Trouble Log Severity levels and Actions

The best thing about the trouble log was that it was always on. Because it was only a write to memory (fast execution) and used a circular buffer (small memory footprint) it was an agile way to balance resource use with the big advantage that you could troubleshoot a bug without altering the way the code executes by enabling the logging system.

When the product was in field test, we could look at a trouble log dump and know a great deal about the state of the system. It was a tremendous help in diagnosing field problems since the science behind the product idea was also being tested. In the development lab, it was wonderful to have the code simply halt when an assertion failed; the whole story was there in the call stack.

Most of the time the problem cause was instantly clear.

This technique depends on the team to put enough calls into the code and at the right places. It also requires that everyone understand the severity system and use it correctly. We relied on code reviews and pair programming to help us achieve thorough, consistent use of trouble log calls in the code.

3.2. Dual-Targeting

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!

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. A #define was used to bracket code that depended on hardware, and skip over it when running on a PC. If the skipped code brought in sensor data, we'd just substitute dummy data when running on the PC. For testing, that was better than real data anyway because we could contrive it to exercise desired code paths.

We would've been overwhelmed without the ability to quickly isolate hardware problems. By gaining confidence in our code through testing it on stable hardware (inside a desktop PC), we could then bring it to the target CPU knowing that the only areas left to test were timing-related behaviors and direct interaction with hardware, such as controlling a motor or LED. Another advantage of this approach is that test data inputs were right there in the code, bracketed by #defines for execution on a PC. Even after the code was migrated to the target CPU, it could be exercised at any time on the PC.

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. Each night we were able to run scripts that built and ran the full bevy of unit tests on the PC, and built the test harnesses for the embedded target, which we could run manually the next day if desired.

3.3. Hardware Driver Unit Tests

The problem in our dual-targeting strategy is that we had no explicit test code for the parts of the software that drove hardware; we were jumping over those spots (as described in the previous section).

In a complex embedded system there will be classes that touch the hardware and others that do not. Of the classes that touch hardware, we must further distinguish between components that are driven by the environment (such as ISRs), and those that drive hardware. For the former, the unit testing strategy is to stub, as described above. But for the latter, we needed a special type of unit test that can only be run on target hardware.

For components that drove hardware, the second unit test was for manual execution only on the target CPU. These hardware unit tests call production routines in the module that directly access hardware to actuate a motor, turn on a LED, or whatever. As an example, our hardware had a 3-position shutter to let all the light signal in, block it completely, or pass it through a filter. The software unit test (the one running on a PC) would skip over code that actuated the shutter motor.

Without our hardware unit test, the only way to check the motor logic would be with the full system running. But it was time-consuming to bring the full system into each of the states that would move the shutter to all its possible positions. An integration test is not a good substitute for a missing unit test. All we really needed was to pass shutter position commands to the function that monitored and operated the motor. That is all the hardware unit test code did.

One could compile that hardware unit test code, load it to target hardware, then step through with the debugger and watch the shutter move to each of its positions.

These hardware unit tests are very valuable for groups outside the software team – electrical engineers, test personnel, and production technicians. For them, the alternative of having to run the full system and trick it into doing things to move the shutter's position was simply not feasible. Ditto for other hardware devices in the system. This kind of support to related professional organizations fostered a collaborative environment between them and the software team, which became exceptionally meaningful during a period of intense phenomenological investigation just prior to going to market.

We put a set of our hardware unit tests together underneath a simple command line menu and sent that to our board vendor to run as a final test before they shipped boards to us. Because the very same code was driving the hardware in the application, we never had to debug code at that level once it correctly drove the hardware. This helped the electrical engineers we worked with to have confidence in our software.

3.4. Domain Level Tests

A specific problem for us was that the serial communications domain was having unexpected delays between receiving a command and responding. Unit tests were too fine-grained to address this, but testing with the full system brought in too many variables. Unit tests could address logic but not timing.

Anytime you cannot distinguish whether a problem is within a domain or is somewhere else in the system, you need domain level tests. In embedded systems you are more likely to run into this kind of situation because of timing constraints.

We could build an executable containing only the serial communications domain, load that to target hardware, and send it test commands over the serial link. From a laptop we could then send commands to the system, which now consisted only of the serial communications domain, and check the response time. Because domains had little coupling to the rest of the system, it was easy to build mock objects to cover those inputs and outputs.

Our timing problem was observed both with the full system running, and with just the serial communications domain running "stand alone". Therefore we knew the problem was within the serial communications domain.

Domains are to the system what methods are to a class. This technique let us use the same testing strategy, raised up one level.

This domain-level testing became a crucial tool for isolating problems observed at the system level, such as timing delays. It was especially helpful when debugging problems associated with the puzzling displays. A distributed team at a partner company developed code for a separate control and display unit for use with our spectrometer. Domain level tests of our team's interfacing domain quickly determined where the problems were.

3.5. Special Test Mode for Volatile Domains

The mathematical algorithm to be implemented in our software was in a preliminary state. Mathematicians were busy revamping it all the time, using Matlab to run test data through its logic. It was a given that the minute we coded it, we'd have stacks of changes to implement, and they'd keep coming.

To keep algorithm changes from becoming a black hole in our efforts, Nancy proposed a plan to have the computer check our computations against data obtained from Matlab by the mathematicians. We did not write unit tests of each stage of computation because preparation of realistic test data (arrays and matrices) would have been difficult to do on our own and the mathematicians' time was already stretched thin.

Instead we took advantage of the work they already did to prep test data for use in Matlab. The data path for our product originated in sensor hardware and went through a variable number of computation stages before a final result was complete. If we could get our software to optionally inject the same initial test data, and compare the end result with the Matlab result, that would make testing much easier. If we got a different end result for the same input data, we'd know there was a mistake in the software. But that

would leave us with the job of narrowing down which stage of the computation had the bug. No problem – it wasn't much more effort to also check the intermediate stage results. The figure below shows this idea.

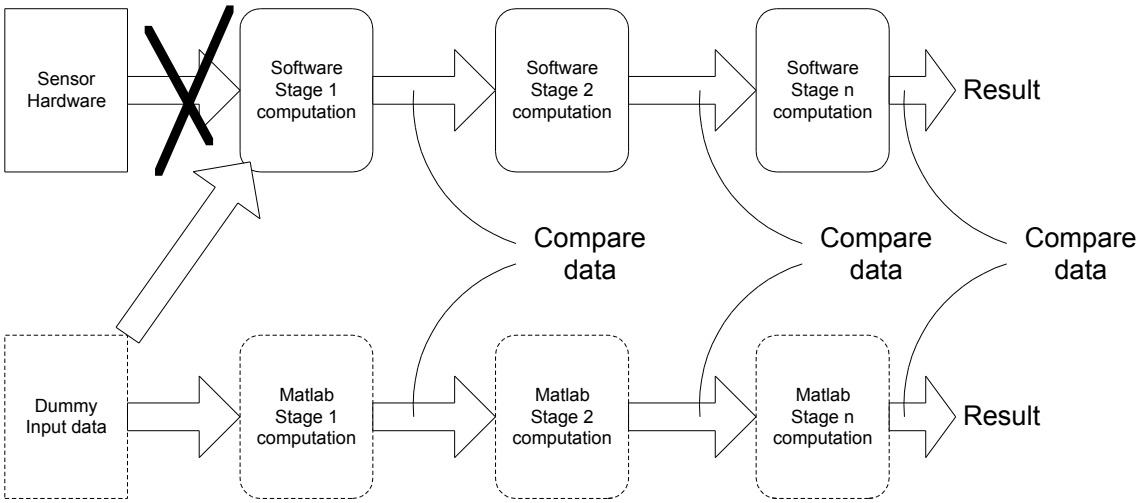

Figure 3. Test Strategy Mathematical Algorithm

The algorithm designers didn't mind giving us the sets of Matlab data from their tests because this was easy to generate. Each time they wanted a new version of the algorithm implemented in the embedded software, they gave us their test data in addition to the math changes. Our code had a variable to enable this test mode, and a set of files to pull the Matlab data from, for comparison. The test mode was conditionally compiled, so the extra memory necessary for this only had to be present on development units.

This technique allowed us to easily implement and test changes no matter how often they occurred. Because the test is a full-system test on target hardware, a long list of possible trouble sources could be checked off quickly.

This technique amounted to doing integration test and unit test of the algorithm simultaneously. The effort to set up the data files and the code to control the test mode was paid back many times over.

In cases where a domain is expected to undergo many changes throughout development, there is high interest by system designers in monitoring the changes at whatever level of maturity the remainder of the system may be in. Iterative releases support concurrent development [3]. In our case, iterations were a great advantage for our cooperation with the algorithm designers. Nevertheless, a domain by nature is not totally independent from other domains, and the effect of other domains may be of interest to system designers.

4. Reflections

The biggest result of using these testing techniques is that we had an amazingly low bug rate. At the unit level, bugs were caught so quickly and easily that we didn't bother to count them. We did keep a log of every bug that made it to integration test or further – those had something to teach us. In three years of development there were only fifty-odd bugs in this group. Our list of open bugs never had more than 2 items at any one time. That means we spent close to 100% of our time adding value instead of scrambling to fix defects.

Our relationship with the hardware group was qualitatively different from what we've observed in non-agile projects. In most shops whenever it's unclear whether a bug is in the software or the hardware, software is guilty until proven innocent. We turned that around by making it very easy to check the software in nicely isolated chunks. The

electrical engineers working with us began to double-check hardware before concluding there was a software problem.

The ability to thoroughly test the math algorithm, and to follow the state of the system out in field tests via dumping the trouble log, allowed us to provide answers when the units behaved strangely in the field and questions came up. New hardware and new spectroscopy science were being tested – it turned out that the software was the most stable part of the system.

5. Acknowledgements

Ellen Lord, Dana Sawyer for their leadership.

Mark Obremski and Mike Grodin, our hardware co-workers, for excellent collaboration.

Bill Grove, Tony Lee, Stephanie Leong for great software teamwork.

6. References

[1] Deming, W. Edwards, *The New Economics For Industry, Government & Education* 2nd Edition, MIT Press, Cambridge, MA, 1993

[2] Maguire, Steve, *Writing Solid Code,* Microsoft Press, 1993

[3] Poppendieck, Mary, *Lean Software Development*, Addison Wesley, 2003