Test Driven Development of Embedded Systems Using Existing Software Test Infrastructure

Micah Dowty
University of Colorado at Boulder
micah@navi.cx

March 26, 2004

Abstract

Traditional software development methodologies mirror the development cycles used in other forms of engineering. Specifications are gathered, software is designed, the design is 'constructed' into a finished software product, then the finished product is tested. New methodologies such as Extreme Programming are improving on this by detecting problems early and flexing under changing requirements.

We focus on applying Extreme Programming's test driven development to embedded systems featuring custom hardware and software designs. An inexpensive and effective method for testing embedded systems using existing software test frameworks and methodology is developed, applied, and evaluated.

1 Introduction

Traditionally, software development mirrors the engineering practices common in older disciplines such as mechanical or electrical engineering. Specifications were gathered from the customer, the specifications were converted to a design, then the design was 'constructed' into a working piece of software. In the 1980s, software engineers began to see the flaws in this method: requirements change often and the 'construction' process isn't nearly as predictable as the construction of a highway or an amplifier.

Several new 'agile' software methodologies were invented to try to combat this problem. Many of these methods focus on acknowledging that software development is an unpredictable process that requires creativity throughout. Extreme Programming is one such methodology.

Traditional software development methodologies divide a project into specification, design, implementation, and testing. A fully working product is not

available until the very end, so there's a good chance it's not at all what the customer really needs, and serious problems will be hidden until testing is underway.

Extreme Programming, however, is based largely on test driven development. A developer writes failing test cases for some bit of necessary functionality, then writes, debugs, and refactors as necessary until 100% of the test cases pass. There always exists a known-working copy of the software with a subset of the final features, and programmers are free to refactor as necessary without creating hidden bugs.

The technique developed in this paper applies some of the lessons learned in software development to the development of hardware and software for embedded systems. We will refer only to embedded systems throughout the rest of this paper, however the same techniques should be equally applicable to other 'soft' hardware systems, such as those based on Field Programmable Gate Arrays.

2 The Problem

Embedded systems are usually built from a diverse set of custom components. This includes conventional software, firmware for very resource-constrained microcontrollers, microcode, FPGA or ASIC designs, and board-level designs.

Software on medium to large platforms may be tested using a conventional test framework, but the interface to other system components must be simulated. Software on very small CPUs or for devices with uncommon architectures often falls outside the capabilities of existing test frameworks. FPGA, ASIC, and board-level designs can be simulated using various software tools, but in isolation from any other system components. Finished PCBs can be tested using specialized electrical testing equipment, but this can be expensive and hard to use during the development cycle.

These conventional tools have their advantages. If, for example, a problem is known to exist in the microcontroller firmware, the firmware can be debugged using a simulator. These tools fall short of a complete solution, however, in some of the same areas where traditional software development methodologies fail. Integration is performed infrequently, problems are hidden until late in development, and developers have little reassurance that the system will still function after refactoring a design, fixing bugs, or adding features.

This problem is exacerbated by the fact that proper test equipment is often quite expensive, so these traditional embedded system development and test techniques can be impossible for hobbyists or projects on a very low budget. In these environments, integration testing is usually performed manually and only inexpensive software simulator tools are available.

What's needed is a test and development strategy that is easy and cheap to apply, detects problems early, and easily adapts to changing requirements.

3 Test Driven Development

To solve some of these problems, we will apply Extreme Programming's test driven development methods to embedded systems with both custom hardware and custom software. Specialized simulator and test systems are not abandoned completely– when available, they are still useful for tracking down a known problem. This solution focuses, however, on incrementally developing and testing a fully integrated system inexpensively using widely available hardware and software.

To make use of existing software testing infrastructure and pose the problem in a simpler way, we encapsulate the embedded system under test in a software layer. Once all the required functionality of the embedded system is represented by this software layer, the software layer can be tested using any compatible off-the-shelf test framework.

This brings several of the same advantages to embedded system development that test driven development brings to software development. The system is working and fully integrated quickly, but without all features. Unit tests reassure the developers that a system is working properly even after extensive refactoring. Test driven development even has advantages unique to embedded systems: software rarely stops working for no apparent reason, but a comprehensive and easy to use test suite for hardware will detect component failures, loose connections, or improper setup quickly and easily.

There are several concerns in using software test infrastructure for hardware, however. The software encapsulation must preserve all of the hardware's likely failure modes, and the test suite must account for these extra failure modes. For example, when testing software it is safe to assume than assigning a value to a variable will always succeed. When testing embedded systems, this may not be so. Timing errors, loose connections, or faulty components could cause the write operation to fail completely or partially.

4 Case Study

Using the above method, test driven development was used to create the 'Critical Decoder' board for the Colorado Space Grant Consortium's Citizen Explorer satellite. The board in question was started late in the satellite's development to replace a faulty earlier design. The previous board had numerous problems that were not discovered until a late integration phase. No conventional electrical test hardware was available, and no unit testing framework was available for the board's CPU, so testing was performed manually and infrequently. To avoid the problems faced by the earlier design, a test driven development strategy was adopted for the new one.

The Critical Decoder board is responsible for several low-level functions of

the satellite that must remain under control even if the main flight computer experiences problems. This includes controlling power to the flight computer and other microcontrollers, routing commands between the satellite's RS-485 network and the radio when the flight computer isn't available, reading several analog sensor values, monitoring deployment status, and controlling radio transmitter power.

5 Test Fixture

Encapsulating the Critical Decoder as a software component required interfacing the board's serial, analog, and digital I/O signals to the computer that would be running the test suite. This would require, on the computer's side, a total of two RS-232 serial ports, one RS-485 serial port, one synchronous serial receiver, 17 analog outputs, 5 digital inputs, and 2 digital outputs. Ideally this interface would be accomplished using an off-the-shelf data acquisition and control board capable of precisely recording and generating test signals.

On the project's limited budget, however, a simpler solution was needed. For this we designed the RCPOD,[1] a simple USB-attached acquisition and control board. The RCPOD includes built-in support for RS-485 and 8 analog input channels with 8-bit resolution. Any of the 30 I/O pins not used for RS-485 or analog input can be used as general purpose bidirectional digital I/O. The RCPOD offers far more I/O for the price than commercial solutions, but its USB 1.1 low-speed interface is much slower than the PCI or full-speed USB interfaces used by commercial data acquisition and control systems. This is only a minor disadvantage, as most of the Critical Decoder's I/O is very low-speed, and the running time of the test suite isn't particularly important. It does raise problems when using the RCPOD with synchronous serial signals or short pulses, as explained later. The RCPOD only provides one RS-485 serial port, so the two required RS-232 ports were supplied by off-the-shelf USB to RS-232 adaptors.

The RCPOD provides a Python[2] interface to its I/O functionality. I/O pins are represented as objects, with methods to test their state, change their state, and change their direction. To create a software encapsulation of the Critical Decoder, we build on this with a Python package for the satellite's command protocol, used to communicate with the Critical Decoder over its RS-485 and RS-232 interfaces. Commands implemented by the Critical Decoder firmware are represented as objects in this package. These objects can be called to carry out the corresponding command. Parameters are automatically serialized into the proper packets, and the response is automatically converted to the most usable representation possible. Errors are reported by raising exceptions. To make the test suite itself as simple and readable as possible, it is important for

[1] Source code and schematics are available at http://navi.cx/svn/misc/trunk/rcpod

[2] An easy to use interpreted object-oriented programming language, see http://python.org

this high-level interface package to be intuitive.

To build the test suite itself, the RCPOD's Python interface and the higher-level command package are used together to form a software encapsulation of the Critical Decoder's hardware. Test cases are written using PyUnit,[3] the standard test framework for Python. PyUnit is based on the proven architecture of JUnit, the standard test framework for Java written by Erich Gamma and Kent Beck. This illustrates another advantage of encapsulating the entire embedded system as a software component: even though the Critical Decoder's firmware is written in assembly language for a processor too tiny to host a test framework, the software encapsulation and test suite can be written in any convenient language.

6 Development

Following the test driven design methodology as closely as practical, test cases were added before or shortly after implementing the hardware or software they test. Much of the Critical Decoder's functionality can be tested by issuing a series of commands and validating the responses. Writing these test cases can be approached in the same way as writing test cases for a pure software system: the test case may report a failure in the presence of working software due to a hardware error, but there is no likely way that the test case could report success in the presence of faulty software and faulty hardware.

Other test cases, however, must focus on the hardware and on interconnection between components. These test cases must be designed carefully in order to cover hardware's unique failure modes. Consider a simplified example in which the system under test is a digital buffer. The test case must ensure that one digital input on the data acquisition and control board always receives the value being transmitted to a digital output. A first attempt at such a test case might set the output to zero, test that the input is zero, set the output to one, then test that the input is one. This test case however could not reliably detect a faulty N-channel transistor on a CMOS buffer. A faulty buffer of this sort would drive a one properly, but its output would be high-impedance when its input is zero. Due to capacitance in the circuit, the value the input would see when the output is low would depend on initial conditions not measurable by the test suite. A more robust test case for this wire would try sending a zero, a one, then another zero. If the output pin has failed in the way described, the first received bit would be undefined but the next two bits would both be one, due to capacitance in the input pin.

Digital inputs and outputs were tested using alternating bit patterns as described above. Additionally, some tests were run first with nearby I/O lines set to zero and repeated with nearby I/O lines set to one, to make it possible to detect short circuits between pins. Ideally every combination of outputs would be tested, but considering the low-speed RS-485 network all commands

[3] http://pyunit.sourceforge.net

must pass through, this would be impractical. The Critical Decoder's watchdog output, analog inputs, and synchronous serial output posed more of a challenge due to limitations of our low-budget RCPOD hardware.

Most of the Critical Decoder's outputs change slowly and only as a response to some command or timer. It has an external watchdog output, however, that emits pulses when PPP packets are detected on the radio's serial port. The Critical Decoder includes a pulse stretcher in software that emits a 13 millisecond pulse when a PPP packet is detected, and ensures that there is at least 13 milliseconds of space between pulses. Instead of trying to accurately sample the pulse shape, the output is sampled repeatedly to estimate its duty cycle. When no PPP packets are present the duty cycle should be 0%, and when PPP packets are being sent continuously it should be 50%. The test case allows a margin of error that should be set according to the expected duty cycle sampling accuracy.

The Critical Decoder's analog inputs pose another problem. Ideally we would test its 17 analog inputs using 17 analog outputs on the RCPOD, but the RCPOD has no analog output capability. While it does require a full D/A converter to characterize the linearity of the Critical Decoder's A/D converter, corner cases can be tested using much simpler hardware. The test fixture for the 17 analog inputs uses 6 of the RCPOD's digital outputs. Pairs of digital outputs are connected to resistor voltage dividers, yielding three analog outputs capable of generating approximately 0, 2.5, or 5 volts. These three analog outputs are then repeated across all 17 analog inputs. This gives a dramatic reduction in test fixture resources required, but still covers all common failure modes including shorts or disconnects on the inputs themselves or on the Critical Decoder's analog multiplexer's address inputs. The repeating pattern of three outputs never ends cleanly on a power of two boundary, so a problem with a multiplexer address line would always 'fold' the inputs in a way that would change the pattern.

The Critical Decoder includes one more output that changes too quickly for the RCPOD to sample, a synchronous serial port used to set the value of a D/A converter that changes the radio's transmit power. The original plan was to connect one of the RCPOD's analog inputs to the output of this D/A converter, but the converter itself is in the satellite's radio rather than in the Critical Decoder, and no spares were available. In place of the D/A converter, a suitable latched shift register circuit was attached to the serial port. The parallel output of this circuit will always match the value the D/A converter would have been set to. For economy of I/O pins, only bits representing corner cases were connected to the RCPOD's inputs. Of the 12 bits available, only bits 0, 7, 8, and 11 were used. This would be enough to ensure the bit pattern was not rotated or truncated, as would be expected if the serial output code failed.

7 Conclusions

Test driven development has been extremely successful for the Critical Decoder project. As the rest of the satellite was being tested during the development of the board, it was helpful to always have a working prototype with a subset of the final functionality. The test suite offered continuous reassurance that the Critical Decoder was functioning properly, which was especially important when it was being used as test equipment for other components in the satellite. Most importantly though, using test driven development ensured that test procedures were always up to date, something that was foreign to us in dealing with the old design.

Test driven development makes an effective and efficient development strategy for embedded systems. For projects on a low budget, good testing can be had using inexpensive data acquisition and control systems and free test frameworks rather than expensive specialized equipment.

Test driven development applied to embedded systems yields many of the same benefits it does in software development, with a few unexpected ones. In prototyping the Critical Decoder, the test suite would immediately notice a broken wire. This increased confidence when making changes to a breadboard covered in wires, as there was no risk in creating hidden problems when rearranging components or transporting the prototype.

8 Future Work

Test driven development using the software encapsulation method is best applied with a high quality data acquisition and control system, but for those on a budget it should be possible to create an enhanced RCPOD or RCPOD-like board that eliminates the problems encountered using it to test the Critical Decoder.

The specific problems encountered could be fixed by adding pulse-measuring functionality and synchronous serial to the RCPOD's firmware. A more general solution would be to create a way to compile timing-critical test cases into a form that could execute directly on the RCPOD. The development of an inexpensive test system with this functionality could make test driven development of embedded systems much more common.

In the Critical Decoder project, a breadboarded interface between the Critical Decoder and the RCPOD had to be constructed. Ideally a test interface would be equipped with a large number of all-purpose I/O pins and simple physical adaptors to connect it to different types of connectors. All electrical routing could be done automatically using multiplexing or an FPGA. This would decrease setup time and increase longevity of a test fixture by eliminating custom hardware.