Effective Test Driven Development for Embedded Software

Michael J. Karlesky, William I. Bereza, and Carl B. Erickson, PhD

Abstract—**Methodologies for effectively managing software development risk and producing quality software are taking hold in the IT industry. However, similar practices for embedded systems, particularly in resource constrained systems, have not yet become prevalent. Today, quality in embedded software is generally tied to platform-specific testing tools geared towards debugging. We present here an integrated collection of concrete concepts and practices that are decoupled from platform-specific tools. In fact, our approach drives the actual design of embedded software. These strategies yield good design, systems that are testable under automation, and a significant reduction in software flaws. Examples from an 8 bit system with 16k of program memory and 255 bytes of RAM illustrate these ideas.**

Index Terms—**Design for testability, Microprogramming, Software quality, Software testing**

I. Introduction

SPECTACULAR software failures make headlines. The problems plaguing Denver International Airport's automated baggage-handling system in the mid-90's had enough media presence to make a Hollywood star jealous. Software bugs with far less media attention are a regularly occurring reality. Bugs derail business plans, exasperate managers and developers, and adversely impact the bottom lines of companies of all sizes.

Embedded software is a unique specialty within the broader software field. High-level IT systems generally run in clean environments and have little contact with the physical world. While bugs are always costly, bugs in PC software or large enterprise applications can be patched with relative ease. In contrast, flaws within the embedded software of an automobile fuel injection system can cause a massive and expensive recall. Worse still is the very real prospect of loss of human life due to embedded software flaws. The abilities, complexities, and pervasiveness of embedded systems continue to grow thus growing the possibility and probability of truly expensive software flaws.

Effective methodologies for managing software risk and producing quality software are beginning to take root in industry. For instance, the practices under the umbrella of "Agile Methodologies" are winning converts [1]. Widespread anecdotal evidence and our own experience testify to "Agile" practices producing an order of magnitude or more reduction in bugs over traditional methods. In particular, among these practices, Test Driven Development (TDD) stands out. TDD is counterintuitive; it prescribes that test code be programmed *before* the functional code those tests exercise is implemented. Practicing TDD means designing software such that it can be tested at any time under automation. Designing for testability in TDD is a higher calling than designing "good" code because testable code *is* good code.

Traditional testing strategies rarely impact the design of production code, are onerous for developers and testers, and often leave testing to the end of a project where budget and time constraints threaten thorough testing. Test Driven Development systematically inverts these patterns.

Practicing TDD follows these essential steps:

1. Identify a piece of system functionality to implement (a single function or method).
2. Program a test to verify that functionality.
3. Stub out the functional code under test (to allow the test code to compile).
4. Compile; run the test and see it fail.
5. Flesh out the functional code.
6. Compile; run the test.
7. Refactor the functional code.
8. Repeat 6 & 7 until the test passes and the functional code is cleanly implemented.
9. Repeat 1-8 until all features are implemented.

In this paper, we draw from experience with resource-constrained systems that do not enjoy the "luxury" of an operating system or object-oriented language (e.g. C++ or Java). Within this context, practicing TDD has generally been regarded as prohibitively difficult. The direct interaction of programming and hardware as well as limited resources for running test frameworks seem to set a hurdle too high to clear. We shall demonstrate the application of a new software design pattern and a multi-tier strategy for testing that brings the efficacy of TDD to even the lowest-level embedded software (and by extension any embedded system).

While the importance of testing in embedded software is universally recognized, testing approaches, in general, have been tied to specific tools or platforms [2]. What we offer here are concepts decoupled from particular tools and platforms.

II. Current State of Testing in Embedded Software & Shortcomings

A. Ad-hoc Testing

Experimentation and ad-hoc testing are often performed

Manuscript submitted April 2006. This paper is based on ideas developed and work performed for various clients as part of ongoing contract software development engagements.

Michael J. Karlesky, William I. Bereza, and Carl B. Erickson, PhD are all with Atomic Object LLC, 941 Wealthy Street SE, Grand Rapids, MI 49506 USA. voice: (616) 776-6020 fax: (616) 776-6015 email: karlesky@atomicobject.com.

during the development process to discover the idiosyncrasies of the system under development. The knowledge gained in these efforts is then applied in the functional source code.

With ad-hoc testing, test fixtures and experimentation code used to characterize the system and shape the functional code are usually discarded or shelved. Over time, these resources fall out of step with system development (or no longer exist at all) and become vestigial remnants of the system's evolution. Valuable, executable knowledge (in the form of code) is lost. Time will almost certainly be lost in later stages of development because these tests have not been kept current.

B. Debugging

Embedded software relies far more heavily on specialized debugging and system inspection tools than does high-level software. Most, if not all, of the need for these tools is created by the multivariable equation of hardware and software commingling. Bugs may be due to hardware, software, or both. Thus, finding the source of unintended behavior generally requires more effort than in high-level software.

The existence of sophisticated debugging tools in embedded software creates an interesting side effect. With such advanced debugging tools available (and needed), developers are inclined to design only for debugging and not for true testing. The assumption inherent in design-for-debug is that any and all code is "debuggable." The limitations in this assumption are threefold. First of all, undesired behaviors in a system under development can be due to any number of obscure reasons – often in the least expected places. Relying on design-for-debug is having faith that one is well-capable of finding a needle in a haystack. Secondly, debugging sessions are one-time events. After a bug is found and corrected there is nothing in place to watch that same code and point out undesired code interaction in the future. Finally, relying heavily on debugging rarely enforces good coding practices; debugging can act as a psychological safety net.

C. Final Testing

The traditional "Waterfall" method of software development prescribes a progression of design, build, and test steps. Final testing is planned as the last major stage of development and verification before release to production. Embedded projects, just as high-level software projects, most often follow these same steps.

Testing planned for the conclusion of a project presents two problems. First of all, time constraints and budget limitations usually squeeze final testing into a compressed time period or eliminate it entirely. As such, tests that might prevent costly future problems are sacrificed for the demands of the present day. Secondly, with testing so removed from development, source code is unlikely to have been developed for ease of testing. For example, a simple temperature measurement might be implemented such that a code block contains both an analog-to-digital conversion and the math routines that will produce a final temperature value. On the surface, there is nothing wrong with this approach. In final testing, however, the math of the routine can only be tested by subjecting the entire system to actual temperature variations or by using a special voltage simulation rig. These tests are not necessarily conclusive from an accuracy standpoint and require elaborate test apparatuses. While simulation environments are certainly necessary for aspects of system testing, a temperature chamber is not necessary to verify five lines of math code.

III. Test Driven Development

A. Overview

Test Driven Development inverts the traditional software development/test cycle. In TDD, the development cycle is not a progression of writing functional code and then later testing it. Instead, testing drives development. A developer looks for ways to make the system testable, designs accordingly, writes tests and creates testing strategies, and then writes functional code to meet the requirements of the test-spawned design [3].

Testing takes different forms. At the highest levels (e.g. integration and system testing) full automation is unusual. At the lowest level, TDD prescribes fully automated unit testing.

In automated unit testing, a developer *first* writes a unit test (a test that validates correct operation of a single module of source code, for instance, a function or method [4]) and then implements the complementary functional code. With each system feature tackled, unit test code is added to an automated test suite. Full regression tests can take place all the time.

Further higher-level testing will complement these code-level tests. Whether this testing is integration or system testing, it will generally follow the patterns of traditional software verification. Ideally, it will also include some measure of automation.

Automated unit tests catch the majority of bugs at a low level and leave for the human mind difficult testing issues like timing collisions or unexpected sub-system interactions.

TDD provides several clear benefits:

1. Code is always tested.
2. Testing drives the design of the code. As a side effect, the code tends to be improved because of the decoupling necessary to create testable code.
3. The system grows organically as more knowledge of the system is gained.
4. The knowledge of the system is captured in tests; the tests are "living" documentation.
5. Developers can add new features or alter existing code with confidence that automated regression testing will reveal failures and unexpected results.

B. Particular Advantage of TDD in Embedded Software

In the context of embedded software TDD provides a further advantage beyond those already listed. Because of the variability of hardware and software during development, bugs are due to hardware, software, or a combination of the two. With TDD, software bugs can be eliminated to such a degree that it becomes far easier to pinpoint, by process of elimination, the source of unexpected system behavior (i.e. hardware versus software).

IV. Unit Testing in Embedded Software

A. Model-Conductor-Hardware Design Pattern

Design patterns are documented approaches to solving commonly occurring problems in the realm of software development. In high-level languages, a multitude of patterns exist that address common situations in elegant and language-independent ways [5].

When considering the challenges of applying TDD to embedded software, the immediately apparent difficulty is the hardware. If tests are to be run in an automated fashion, then hardware functions must be automated as well. Initially, this seems to be a task too complicated to implement in a cost-effective manner. Simulating hardware events in a comprehensive manner can require complicated platform-specific tools or elaborate programmable hardware test fixtures. However, using existing design patterns as inspiration, another approach is available.

In a Graphical User Interface (GUI), we find a situation akin to embedded software. Namely, in GUI programming there is a tendency to mix functional logic with event handling external to that logic. In this analogy, the asynchronous events and programming interfaces of on-screen widgets are similar to external interrupts and hardware registers of embedded systems.

Software developers have created two patterns to address the problems inherent in architecting good GUI applications. The Model-View-Presenter (MVP) and Model-View-Controller (MVC) design patterns effectively and cleanly separate widget event handling from flow control logic [6]. As a side effect, the separation these patterns provide allows for automated testing of GUI presentation code free from a user clicking upon actual widgets.

In MVP, the View represents a very thin wrapper around a collection of widgets comprising a GUI. The Model saves state external to the widgets and interfaces with programming functions elsewhere in the system not directly related to the GUI. The Presenter references both the Model and View and embodies the presentation logic necessary to process events from the GUI's widgets and change state on those widgets. With this separation, a View can be "mocked", or simulated. A mocked View, under automated test control, can emit events and receive calls from the Presenter. Similarly, the Model can be mocked. In this way, the presentation logic of the Presenter can be thoroughly tested in an automated fashion separate from an on-screen GUI [7]. It is assumed that the widgets of the View are already well-tested by system vendors. Tests for the Presenter and Model are created and added to the automated test suite. Functional code is written to cause the tests to pass. Final verification by an actual user ensures that all has been correctly connected in the production system apart from the test system.

Drawing from MVP and MVC, we developed the Model-Conductor-Hardware (MCH) pattern for use in embedded software. In this pattern, similar to its GUI cousins, MCH allows the physical hardware to be mocked, forces functional logic to be decoupled from hardware, and provides a means for automated unit test suites to test both the hardware and functional logic.

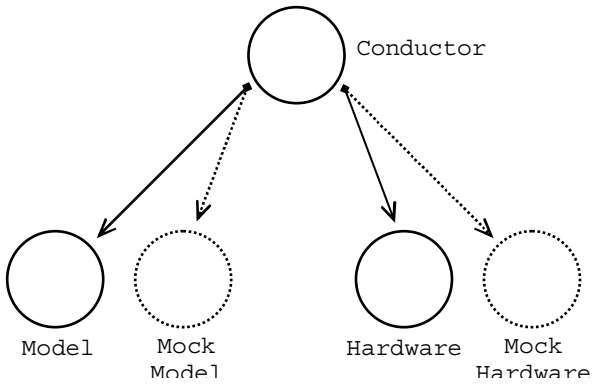

Fig. 1. Relationships of a Conductor to its complementary MCH triad members and their mocks. The depicted mocks stand in for the concrete members of the triad and allow for testing of the logic within the Conductor. A mock Conductor (not depicted) allows the concrete Model and Hardware to be tested. Global variables within mocks capture function parameters and simulate return values used in test assertions.

Model

The Model in MCH models the current state of the system. For example, if an analog output is set to +5V, but the feedback circuit reports +4V (with tolerance of 100mV), then the Model will set a corresponding error state within itself for use by the Conductor. The Model ensures internal consistency of states in this manner. The Model is only connected to the Conductor and has no direct reference to the Hardware.

Hardware

The Hardware in MCH represents a thin layer around the hardware itself. This member of the MCH triad encapsulates the ports and registers used in the system. Interrupt Service Routines (ISR's) notify the Conductor of system state changes. The Hardware is only connected to the Conductor and has no direct reference to the Model.

Conductor

The Conductor in MCH contains the main control logic of the triad. The Conductor is triggered by the Hardware to process new data and events. Upon such triggers, the Conductor sets the state within the Model and uses the state contained by the Model in its logic to send commands or set data in the Hardware. The Conductor contains a control loop and acts as the intermediary between the Model and the Hardware. The Conductor was so named because of its role as a system director and because of its proximity to actual electrical components.

For simple systems a single MCH triad may be sufficient. Often, multiple triads are necessary to simplify the logical segregation of testing. In these cases, triads generally exist independently of one another with a central "executor" to call each Conductor's control loop. If there is any overlap between triads, it tends to happen at the Hardware level.

B. Testing with Model-Conductor-Hardware

Testing with MCH centers on making test assertions against the information captured in mocks. Each functional member of the triad is unit tested in isolation from the system via

composition with mocks of the system. The calls and parameters of the triad member under test are captured within the mocks for test assertions. The proper operation of the logic under test is revealed by its actions on the mocks.

With mocks constructed for each member of the MCH triad, clear testing possibilities become apparent. Code testing via simulator, on-chip, or in a cross-compiled environment are all possible. The states and behavior within the Model are tested independently of hardware events and functional logic. The system logic in the Conductor is tested with simulated events from Hardware and simulated states in the Model. With a mock Conductor, even hardware register configuration code and ISR's can be tested via a simulator, hardware test fixture, or board-level feedback loops. MCH code examples follow in a later section of this paper.

C. Unit Testing Framework

Unit testing frameworks exist for nearly every high-level programming language in common use today. The mechanics of a test framework are relatively simple to implement [8]. A framework holds test code apart from functional code, provides functions for comparing expected and received results from the functional module under test, and collects and reports the test results for the entire test suite.

In our work, we have both customized the open source project Embunit (Embedded Unit) and created a very lightweight framework called Unity [9]. Embunit and Unity are both C-based frameworks we modify for target platforms as needed.

D. The cost of using Model-Conductor-Hardware

MCH adds little to no overhead to a production embedded system. Of course, mocks are not included in the final system. Further, MCH is essentially naming, organization, and calling conventions with little to no extra memory use or function calls; any overhead incurred by these conventions is easily optimized away by the compiler.

TDD, in general, does add to project cost in added developer time. However, clear savings are realized over the system's lifetime in reduced bugs, reduced likelihood of recall, and ease of feature additions and modifications.

V. TDD IN EMBEDDED SOFTWARE

A. Four Tier Testing Strategy

Thorough software testing includes automated unit testing at the lowest level and integration and system testing at higher levels. Unit testing was addressed in the preceding section. Implementing an overall TDD strategy in embedded software is a four tier testing approach. With each step up through the tiers, less automated testing occurs and more human interaction is required. However, each tier provides increasing test confidence and frees developers and testers to use their human intelligence and intuition for difficult testing matters such as sub-system interaction and timing collisions. Automated testing at the lowest levels of system development can eliminate a high number of bugs early on. Ultimately, system flaws found earlier in the development process cost less than those found later.

1) Automated Unit Testing

Developers use MCH to decouple functional logic code from hardware code and develop unit tests to be run in an automated test framework. These tests are run on-chip, cross-compiled on a PC, or executed in a platform simulator such that automated regression tests can always be executed. Note that work in this tier can progress without target hardware.

2) Hardware Level Testing

Developers and engineers use a combination of unit tests, hardware features, and direct developer interaction to test hardware functions and hardware setup code. Using feedback loops designed into the hardware, processor diagnostic functions, hardware test fixtures, and user interaction, all hardware functions are tested. The approach taken here is system-dependent. Once hardware functions are tested and operational, it is likely that tests developed here will be run far less frequently than in Tier 1.

3) Communication Channel Testing

If the embedded system includes an external communication interface, developers use PC tools to exercise and capture test results of the system through this channel. A complementary hardware test fixture, software test fixture, and/or significant human interaction are likely to be required to exercise the system and provoke communication events.

4) End to End System Testing

Having confidence in low level test successes, developers and/or testers manually exercise an end-to-end exploratory system test looking for emergent timing issues, responsiveness deficiencies, UI inconsistencies, etc.

B. Continuous Integration

The technique of continuous integration regularly brings together a system's code (possibly from multiple developers) and ensures via regression tests that new programming has not broken existing programming. Automated build systems allow source code and tests to be compiled and run automatically. These ideas and tools are important supports to effective TDD but are beyond the scope of this paper [10].

VI. EMBEDDED MODEL-CONDUCTOR-HARDWARE EXAMPLES

A. MCH in a C-based Environment

Creating mocks and tests in an embedded C environment is accomplished through compiled `mock.o` implementations of header file function declarations. For example, suppose `hardware.h` declares all functions for interfacing the hardware features of a particular microcontroller. In this example, Conductor tests will verify that the Conductor makes specific calls on the hardware with appropriate parameters. As such, a `mockhardware.c` definition file will be written containing otherwise empty functions that store individual function call parameter values or return specific values – both as defined by global variables. Object files for mock and functional code are linked together, and tests access the previously mentioned global values to verify the Conductor calls to the `mockhardware.h` interface.

B. Code Samples

The following code blocks are examples drawn from a real-

world project developed for Savant Automation of Grand Rapids, Michigan. Savant builds Automated Guided Vehicles. These samples pertain to a dedicated speed control board; the functions shown set output drive voltage. In this testing scenario, we illustrate a Conductor under test. The example tests verify that the Conductor is correctly using the hardware interface (note use of ASSERT macros from test framework).

hardware.h

```
/// Get feedback from analog drive output.
millivolts Hardware_GetFeedbackVoltage(void);

/// Set the drive output voltage
void Hardware_SetOutputVoltage(
  millivolts output);

/// Set the error flag.
void Hardware_SetError(bool err);
```

model.h

```
typedef struct _ModelInstance {
  millivolts FeedbackVoltage;
  millivolts OutputVoltage;
  bool Error;
} ModelInstance;

/// Set the feedback voltage.
void Model_SetFeedbackVoltage(
  millivolts feedback);

/// Get drive output voltage for hardware.
millivolts Model_GetOutputVoltage(void);

/// Get the error state.
bool Model_GetError(void);
```

conductor.h

```
/// Callback for hardware feedback voltage.
void Conductor_HandleFeedbackVoltage(void);

/// Control loop called by main() forever
void Conductor_Run(void);
```

mockhardware.h

```
/// Feedback voltage to return
extern millivolts
  Hardware_InputFeedbackVoltage;

/// Output voltage set by conductor.
extern millivolts
  Hardware_OutputDriveOutputVoltage;

/// Error flag
extern bool Hardware_OutputError;
```

mockhardware.c

```
#include "mockhardware.h"
#include "hardware.h"
#include "conductor.h"

millivolts Hardware_InputFeedbackVoltage;
millivolts Hardware_OutputDriveOutputVoltage;
bool Hardware_OutputError;

millivolts Hardware_GetFeedbackVoltage(void) {
  return Hardware_InputFeedbackVoltage;
}

void Hardware_SetOutputVoltage(
  millivolts output){
  Hardware_OutputDriveOutputVoltage = output;
}

void Hardware_SetError(bool err) {
  Hardware_OutputError = err;
}
```

mockmodel.h

```
/// Modeled system feedback voltage
extern millivolts Model_FeedbackVoltage;

/// Modeled system output voltage
extern millovolts Model_OutputVoltage;

/// Modeled error state
extern bool Model_Error;
```

mockmodel.c

```
// Linked with testconductor.o in place of
// model.o to allow conductor tests
// independent of logic in actual model.

#include "mockmodel.h"

millivolts Model_FeedbackVoltage;
millovolts Model_OutputVoltage;
bool Model_Error;

void Model_SetFeedbackVoltage(
  millivolts feedback) {

  Model_FeedbackVoltage = feedback;
}

millivolts Model_GetOutputVoltage(void) {
  return Model_OutputVoltage;
}

bool Model_GetError(void) {
  return Model_Error;
}
```

model.c

```
#include "model.h"

ModelInstance Model;

void Model_SetFeedbackVoltage(
  millivolts feedback) {

  Model.FeedbackVoltage = feedback;

  if(feedback != Model.OutputVoltage) {
    // realistically use nominal value
    Model.Error = true;
  }
}

millivolts Model_GetOutputVoltage(void) {
  return Model.OutputVoltage;
}

bool Model_GetError(void) {
  return Model.Error;
}
```

conductor.c

```
#include "model.h"
#include "hardware.h"

void Conductor_HandleFeedbackVoltage(void) {
  Model_SetFeedbackVoltage(
    Hardware_GetFeedbackVoltage());
}

void Conductor_Run(void) {
  Hardware_SetError(Model_GetError());

  if(!Model_GetError()) {
    Hardware_SetOutputVoltage(
      Model_GetOutputVoltage());
  }
}
```

testconductor.c

```
#include "conductor.h"
#include "mockhardware.h"
#include "mockmodel.h"

static void testHandleFeedback(void) {
  Hardware_InputFeedbackVoltage = 7;

  Conductor_HandleFeedbackVoltage();

  TEST_ASSERT_EQUAL_INT( 7,
    Model_FeedbackVoltage);
}

static void testConductorRun(void) {
  Model_Error = false;
  Model_OutputVoltage = 78;

  Conductor_Run();

  TEST_ASSERT_MESSAGE(
    Hardware_OutputError == false,
    "Error set incorrectly");

  TEST_ASSERT_EQUAL_INT( 78,
    Hardware_OutputDriveOutputVoltage);

  Model_Error = true;
  Model_OutputVoltage = 99;

  Conductor_Run();

  TEST_ASSERT_MESSAGE(
    Hardware_OutputError == true,
    "Error not set");

  TEST_ASSERT_EQUAL_INT(78,
    Hardware_OutputDriveOutputVoltage);
}
```

VII. Conclusion / Future Work

Applying Test Driven Development in embedded software allows developers to create well-tested systems. The concepts we have presented are tool and platform independent allowing the methods of TDD to drive design and test implementation. Software design and quality are both improved leading to overall cost savings in reduced field defects and eventual feature enhancements. TDD and to a lesser extent initial setup and tool customization for each new project will add to project development time; however, the benefits far outweigh this cost. Total test automation can likely never be accomplished. Nevertheless, the presented methods codify a flexible approach that encourages consistent testability.

Planned future work includes applying and expanding the practices covered in this paper in the context of Real Time Operating Systems. Further, given our experience with our own Unity framework, we believe that it is feasible to create a unit test framework for assembly languages.

Acknowledgment

Authors thank Matt Werner of Savant Automation for the opportunity to implement in a production system the ideas that inspired this paper.

References

[1] Kent Beck, *Extreme Programming Explained*, Reading, MA: Addison Wesley, 2000.
[2] Wolfgang Schmitt. "Automated Unit Testing of Embedded ARM Applications." *Information Quarterly*, Volume 3, Number 4, p. 29, 2004.
[3] David Astels, *Test Driven Development: A Practical Guide*, Upper Saddle River, NJ: Prentice Hall PTR, 2003.
[4] "Unit Test." http://en.wikipedia.org/wiki/Unit_test.
[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley Professional Computing Series, 1995.
[6] Martin Fowler. "Model View Presenter." http://www.martinfowler.com/eaaDev/ModelViewPresenter.html. July 2004.
[7] M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, C. Stienstra. "Presenter First: Organizing Complex GUI Applications for Test-Driven Development," accepted at Agile 2006 conference, Minneapolis, MN.
[8] Kent Beck, "Simple Smalltalk Testing: With Patterns." http://www.xprogramming.com/testfram.htm.
[9] http://www.atomicobject.com/embeddedtesting.page.
[10] "Continuous Integration." http://en.wikipedia.org/wiki/Continuous_integration.