Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns

Embedded Systems Conference Silicon Valley (San Jose, California)
ESC 413, April 2007

Michael Karlesky
karlesky@atomicobject.com
William Bereza
bereza@atomicobject.com

Greg Williams
williams@atomicobject.com
Matt Fletcher
fletcher@atomicobject.com

Atomic Object, 941 Wealthy Street SE, Grand Rapids, MI 49506
http://atomicobject.com

Abstract – Despite a prevalent industry perception to the contrary, the agile practices of Test-Driven Development and Continuous Integration can be successfully applied to embedded software. We present here a holistic set of practices, platform independent tools, and a new design pattern (Model Conductor Hardware - MCH) that together produce: good design from tests programmed first, logic decoupled from hardware, and systems testable under automation. Ultimately, this approach yields an order of magnitude or more reduction in software flaws, predictable progress, and measurable velocity for data-driven project management. We use the approach discussed herein for real-world production systems and have included a full C-based sample project (using an Atmel AT91SAM7X ARM7) to illustrate it. This example demonstrates transforming requirements into test code, system, integration, and unit tests driving development, daily "micro design" fleshing out a system's architecture, the use of the MCH itself, and the use of mock functions in tests.

Introduction

HEADS around the table nodded, and our small audience listened to us thoughtfully. Atomic Object had been invited to lunch with a potential client. We delivered our standard introduction to Agile[1] software development methods. While it was clear there was not complete buy-in of the presented ideas, most in the room saw value, agreed with the basic premises, or wanted to experiment with the practices. And then we heard an objection we had never heard before: "Boy, guys, it sounds great, but you can't do this with firmware code because it's so close to the hardware." Conversation and questions were replaced with folded arms and furrowed brows. It was the first time we had had in-depth conversation with hardcore embedded software developers.

In standard Atomic Object fashion, we took our experience as a dare to accomplish the seemingly impossible. We went on to apply to embedded software what we knew from experience to be very effective and complementary techniques – in particular Test-Driven Development (TDD) and Continuous Integration (CI). Inspired by an existing design pattern, we discovered and refined an approach to enable automated system tests, integration tests, and unit tests in embedded software and created a small C-based test framework[2]. As we tackled additional embedded projects, we further refined these methods, created a scriptable hardware-based system test fixture, developed the means to auto-generate mock functions for our integration tests, wrote scripts to generate code skeletons for our production code, and tied it all together with an automated build system.

The driving motivation for our approach is eliminating bugs as early as possible and providing predictable development for risk management. We know by experience that the methods we discuss here reduce software flaws by an order of magnitude or more over the average[4]. They also allow a development team to react quickly and effectively to changes in the underlying hardware or system requirements. Further, because technical debt[3] is virtually eliminated, progress can be measured and used in project management. A recent case study of a real-world, 3 year long, Agile embedded project found the team out-performed 95th percentile, "best in class" development teams[4]. Practices such as Test-Driven Development (TDD) and Continuous Integration (CI) are to thank for such results.

Within embedded software circles, the practices of TDD and CI are either unknown or have been regarded as prohibitively difficult to use. 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. Our approach has been successfully applied in systems as small as 8 bit microcontrollers with 256 bytes of RAM and scales up easily to benefit complex, heavy-duty systems.

Application of these principles and techniques does not incur extra cost. Rather, this approach drastically reduces final debugging and verification that often breaks project timelines and budgets. Bugs found early are less costly to correct than those found later. Technical debt is prevented along the way, shifting the time usually necessary for final integration and debugging mysteries to developing well-tested code prior to product release. Because code is well-tested and steadily and predictably added to the system, developers and managers can make informed adjustments to priorities, budgets, timelines, and features well before final release. In avoiding recalls due to defects and producing source code that is easy to maintain and extend (by virtue of test suites), the total software lifecycle is less costly than most if not all projects developed without these practices.

The Value of TDD and CI

Test-Driven Development and Continuous Integration are complementary practices. Code produced test-first tends to be well designed and relatively easy to integrate with other code. Incrementally adding small pieces of a system to a central source code control system ensures the whole system compiles without extensive integration work. Running tests allows developers to find integration problems early as new code is added to the system. An automated build system complemented by regression test suites ensures a system grows responsibly in features and size and exists in a near ready-to-release fashion at all times. For a more thorough discussion of these practices, please see the appendix.

Particular Advantages of TDD and CI in Embedded Software

In the context of embedded software, TDD and CI provide two further advantages beyond those already discussed. First, because of the variability of hardware and software during development, bugs are due to hardware, software, or a combination of the two. TDD and CI promote a strong separation of concerns such that it becomes far easier to pinpoint, by process of elimination, the source of unexpected system behavior. Well-tested software can generally be eliminated from the equation or, in fact, used to identify hardware issues. Second, because these techniques encourage good decoupling of hardware and software, significant development can occur without target hardware.

Russian Dolls & Our Embedded Software Development Approach

A set of Russian dolls comprises individual dolls of decreasing size and corresponding detail nested one inside another. Our approach fleshes out a system's architecture with Russian doll-like levels of test-driven design. The architecture of a system and its requirements drive system tests. System tests in turn drive integration tests. Integration tests drive unit tests. Each nested level of executable testing drives the design of the production code that will satisfy it. Implementing testable code forces a series of small design decisions at the keyboard. The result is a system aggregated of high quality, thoroughly tested pieces nested together to support the overall architecture and satisfy the system requirements.

In this section, we provide background on specialized techniques we employ, discuss the tools supporting our approach, and finally present a summary of the steps to implement a single system feature from start to finish. Our techniques and tools are synergistic; each supports and enhances the others. The paper concludes with an in-depth discussion of the Model Conductor Hardware design pattern introduced in this section and an end-to-end working example with tests, mocks, and source code.

Techniques

System, Integration, and Unit Testing

Requirements are composed of one or more features. We satisfy requirements by implementing features. Each feature necessitates creating a system test that will exercise and verify it once it exists. This system test will operate externally to the system under test. If pressing a button is to generate bytes on a bus, a system test specifies in programming the initiation of the button signal and the verification of the bytes on the bus.

A single system feature is composed of one or more individual functions. We begin creating each of these functions by programming integration tests to verify the interrelation of function calls. Subsequent unit tests verify the output of each function under various input conditions. After creating the integration and unit tests, we run them and see them fail. Next, we write the production code that will satisfy these tests. Tests and source are optionally refactored until the code is clean and the tests pass. Integration and unit tests can be run on target hardware, cross-compiled and run on the development machine, or run in a simulator, depending on the development environment and target system.

Interaction-based Testing & Mock Functions

Few source code functions operate in isolation; most make calls on other functions. The composition of inter-function relationships, in large part, constitutes the implementation of a system feature. Testing this interaction is important. Mock functions facilitate testing these interactions and have become a key component of our development at the integration level. We practice interaction-based testing for integration tests and state-based testing for unit tests[5].

Whenever a function's work requires it to use another complex function (where a complex function is one requiring its own set of tests), we mock out that helper function and test against it. A mock presents the same interface to the code under test as the real module with which it interacts in production. Functionality within the mock allows tests to verify that only the expected calls with expected parameters were made against it in the expected order (and no unexpected calls were made). Further, a mock can produce any return results the tester-developer requires. This means that any scenario, even rare corner cases, can be verified in tests.

Hardware and Logic Decoupling (Model Conductor Hardware design pattern)

Specialized hardware and accompanying programming interacting with it is the single greatest complication in thoroughly testing an embedded system. We discovered our Model Conductor Hardware (MCH) design pattern in the process of segregating and abstracting hardware from logic to enable automated testing. The Model, Conductor, and Hardware components each contain logically related functions and interact with one another according to defined rules. With these abstractions, divisions, and behaviors, the entire system can be unit and integration tested without direct manipulation of hardware. A later section of this paper provides a more in-depth explanation of MCH.

Conductor First

A complete embedded system is composed of multiple groups of Model, Conductor, and Hardware components. A single interrelated group of these components is called an MCH triad. Each triad represents an atomic unit of system functionality. The Conductor member of an MCH triad contains the essential logic that conducts the events and interactions between the Hardware and Model comprising the functionality under test.

Conductor First (inspired by Presenter First[6,16]) is our approach to allow TDD to occur at the embedded software unit and integration level. We start by selecting a piece of system functionality within a system requirement. From this, we write integration and unit tests for the Conductor with a mock Hardware and a mock Model. Production code is then written to satisfy the Conductor tests. This technique allows us to discover the needed Hardware and Model interfaces. The Hardware and Model are then implemented in a similar fashion, beginning with tests; the Hardware and Model tests reveal the needed use of the physical hardware and the interface to other triad Models.

Starting in the Conductor guides development of function calls from the highest levels down to the lowest. Developing the Conductor with mocks allows the control logic necessary to satisfy the system requirement to be designed and tested with no coupling to the hardware or other system functionality. In this way, unnecessary infrastructure is not developed and system requirements are implemented as efficiently and quickly as possible.

Tools

Systir – System Test Framework

Systir[7] stands for "System Testing in Ruby." Ruby[8] is the reflective, dynamic, object-oriented scripting language Systir is built upon and extends. In TDD, we use Systir to introduce input to a system and compare the collected output to that which is expected in system tests. Systir builds on two powerful features of Ruby. First, Systir uses Ruby-based drivers that can easily bind to libraries of other languages providing practically any set of features needed in a system test (e.g. proprietary communication libraries). Second, Systir allows us to create Domain Specific Languages[9] helpful in expressing tests in human readable verbiage. We developed Systir for general system testing needs; it has also proven effective for end-to-end embedded systems testing.

Scriptable Hardware Test Fixture

System testing embedded projects requires simulating the real world. The miniLAB 1008[10] is the device we have adopted as a hardware test fixture. It provides a variety of analog and digital I/O functions well suited to delivering input and collecting output of an embedded system under development. A proprietary library allows a PC to communicate with a miniLAB 1008 via USB. We developed a Ruby wrapper around this library to be used by Systir system tests. Other test hardware and function libraries (e.g. LabWindows/CVI) could also be driven by Systir tests with the inclusion of new Ruby wrappers.

Source, Header, and Test File Code Generation

We decouple hardware from programming through the MCH design pattern. We also utilize interaction-based testing with mock functions. Both of these practices tend to require a greater number of files than development approaches not using MCH and interaction-based testing. At the same time, because of the pattern being used, we have very repeatable (i.e. automation friendly) file creation needs. To simplify the creation of source, header, and test files, we created a Ruby script to generate source, header, and test skeleton files.

Argent-based Active Code Generation

The skeleton files created by our file generation script contain pre-defined function stubs and header file include statements as well as Argent code insertion blocks. Argent[11] is a Ruby-based, text file processing tool that populates tagged blocks with the output of specified Ruby code. Our file generation script places Argent tags in the skeleton files that are later replaced with C unit test and mock function management code necessary for all project test files.

Unity – Unit Test Framework for C

The mechanics of a test framework are relatively simple to implement[12]. A framework holds test code apart from functional code, provides functions for comparing expected and received results from the functional code under test, and collects and reports test results for the entire test suite. Unity[13] is a unit testing framework we developed for the C programming language. We found no good, lightweight testing framework for embedded C so we created our own. We customize Unity reporting per project (e.g. printing results through `stdio`, a serial port, or via simulator output).

CMock – Mock Function Library for C

CMock[13] is a Ruby-based tool we created to automate creation of mock functions for unit testing in the C language. CMock generates mocks from the functions defined in a project's header files. Each mock contains functionality for capturing and comparing calls made on the mock to expectations set in tests. CMock also allows tests to specify return results from functions within the mock. CMock alleviates the pain of creating and maintaining mocks; consequently, developers are motivated to make good design changes, since they need not worry about updating the mocks manually.

Dependency Generator & Link Time Substitution – File Linking Management for C

To facilitate linking of source files, test files, and mocks for testing or release, we developed a Ruby-based tool to manage these relationships automatically. This dependency tool inspects source and header files and assembles a list of files to be linked for testing or release mode.

In an object-oriented language, we would normally compose objects with delegate objects using a form of dependency injection[14]. Because C has no formalized notion of objects, constructor injection cannot be used. Instead, we substitute the CMock generated mock functions for real functions at link time.

Rake – Build Utility

Rake[15] is a freely available build tool written in Ruby ("Ruby make"). We create tasks in Rake files to compile and link a system under development, generate mocks with CMock, run Argent code generation for unit and integration testing, run our Unity unit test framework, and run our Systir system tests.

Subversion & DCI – Code Repository & Automated Build System

We use the freely available source code control system Subversion to manage our project files, source code, and unit and system test files. As tests and source code are implemented and those tests successfully pass, we check our work into Subversion. Upon doing so, an internally developed automated build system, DCI, pulls down the latest version of the project from Subversion, builds it, runs its tests, and reports the results. This process repeats itself upon every check in. Of course, Subversion and DCI can be replaced by a myriad of tools.

Implementing a System Feature – An Overview

Step 1 – Create One or More System Tests

The process begins with creating a system test to verify an aspect of a requirement with a specific test case. Some amount of Systir driver code will be necessary to establish the language of the system test and provide sufficient functionality (e.g. digital I/O, analog voltages, network traffic, etc.) to exercise the hardware of the embedded system. For direct hardware interaction, we make calls to our miniLAB wrapper. For data over communication channels (e.g. RS232 or Ethernet) calls to other libraries are necessary.

A typical system test generates a stimulus input and then gathers output from the board under development. Assertions comparing the expected and collected output will determine whether the system test passes or fails. Of course, at this point in the process, there no production code for this feature exists so this system test will fail until the feature is complete. The constraints of the system test, however, even at this early stage, will guide implementation and force disambiguation of the system requirement currently driving development.

Step 2 – Generate an MCH Triad and Unit Tests

No production code or unit tests yet exist. Development continues with using the file generation script. A Model, Conductor, and Hardware are each generated with a source, header, and unit test skeleton file.

Step 3 – Add calls to the Executor for the Conductor's Init and Run Functions

A system of even modest complexity will contain several MCH triads. For multi-triad systems, a single Executor contains the main execution loop and calls each Conductor to run its logic. Because the order in which calls made on the Conductors is usually significant, the Executor must be edited by hand. Before calls to the new Conductor are added to the Executor, however, the Executor's unit tests are updated to ensure that all the Conductors are being called properly. Within the Executor tests, CMock generated mocks are used to stand in for the actual Conductors.

Step 4 – Create Conductor Unit Tests & Production Code

Starting from the system requirement currently being implemented, we next determine the individual functions necessary to satisfy that requirement. These are expressed as Conductor unit tests. Once the tests are created, production code to satisfy these tests is added to the Conductor.

The logic of the Conductor is tested against a mock Hardware and Model. Each function call made against the Hardware or Model must be added in the Hardware or Model's header file. CMock will automatically generate the corresponding mocks from the Hardware and Model header files prior to running the unit tests.

Step 5 – Create Model & Hardware Unit Tests & Production Code

The Model and Hardware triad members are next implemented. Tests are implemented first; in the process, the interfaces for calls on the physical hardware and any interaction with other triad Models are revealed (interaction among multiple triads is discussed in depth in the MCH section).

Both the Hardware and Model will likely have initialization functions called by the Conductor. Functions in the Hardware and Model (already specified in the Conductor tests) may best be implemented by delegating responsibility to other helpers. In such cases, the module generation script is used to create these helper modules. Tests in the Model and Hardware will make calls against mocks of these helper functions.

The Hardware triad component, of course, makes calls to the physical hardware itself. Unit tests here generally involve modifying registers, calling the functions of the triad's Hardware member, and then making assertions against the resulting register values. This can be complicated by read-only and write-only registers. Simulators often allow violations of these restrictions which may benefit or hinder test development.

Step 6 – Complete All Test and Production Programming

The tests and functional code of the Model, Conductor, Hardware, and helpers may require refactoring and subsequent re-running until the code is clean and all tests pass.

When the unit tests are run, a Rake task will: call CMock to automatically generate Hardware and Model (and helper) mocks from header files, call Argent to insert Unity and CMock functions in appropriate files, use the dependency generation tool to create a list of files to be linked (including the unit tests, mocks, and Unity framework itself), run the compiler and linker, and then run the resulting test executable appropriately. Executing the test executable may require starting a simulator or flashing the target hardware and collecting test results from a serial port or other communication channel.

Once all the unit tests pass, the system can be compiled in non-test mode, loaded onto the hardware, and run against the system tests and hardware test fixture. Changes and fixes may be necessary to cause system tests to pass. Once tests pass, the new and modified files are checked into the source code repository; our automated build system will build and test the entire project and report results. Work on the next feature then begins anew.

Model Conductor Hardware Design Pattern & Conductor First

Design patterns are documented approaches to solving commonly occurring problems in software development. A multitude of patterns exist that address common situations in elegant and language-independent ways[12]. The Model Conductor Hardware pattern decouples functional logic from hardware for testing. With this decoupling, automated regression suites of integration and unit tests can be run on-chip, cross-compiled on a PC, or executed in a platform simulator. Working in this manner can even generate significant progress without target hardware.

Model Conductor Hardware Similarities to Model View Presenter

The basic MCH concepts and naming[*] are modeled on the MVP design pattern. The Model of both patterns serves

[*] The Conductor was so named because of its role as a system director and because of its proximity to actual electrical components. Because of the correspondence to MVP, one could naturally assume that "Model Conductor Hardware" should be called "Model Hardware Conductor." The apparent

the same basic function – to model the system (e.g. mathematical equations, business logic, data access, etc.). The Hardware component of MCH is analogous to the View of MVP – both are lightweight abstractions that allow decoupling of system components and programming logic. The Conductor component of MCH serves a similar purpose to the Presenter of MVP – both define the order and composition of interactions within the triad upon an event occurring in either of the other triad components. Development in MCH starts with tests in the Conductor similar in fashion to the Present First[6,16] method of MVP development.

Model

The Model in MCH models the system (e.g. mathematical equations, control logic, etc.), holds state, and provides a programmatic interface to the portion of the system that exists outside a single MCH triad. The Model is only connected to the Conductor and has no direct reference to the Hardware.

Conductor

The Conductor in MCH conducts the flow of data between the Model and Hardware and is stateless. The Conductor acts when triggered by the Hardware or Model. It captures a system function in Model and Hardware interactions: setting state within the Model, querying the state contained by the Model, querying the state contained in the Hardware, moving data from the Model to the Hardware, moving data from the Hardware to the Model, and initiating the hardware functions encapsulated by the Hardware.

Hardware

The Hardware in MCH represents a thin layer around the physical hardware itself (e.g. ports, registers, etc.) and all its accompanying functions and state-based facilities. The Hardware notifies the Conductor of events by providing functions that check state in the hardware. State can be hardware registers or flags set by Interrupt Service Routines (ISRs) and other Hardware routines. The Hardware is only connected to the Conductor and has no direct reference to the Model.

Unit Testing with Model Conductor Hardware

With mocks constructed for each member of the MCH triad, clear testing possibilities become apparent (please see preceding sections for a discussion of mocks and interaction-based testing). The states and behavior within the Model are tested independently of hardware events and control logic. Each system requirement's logic in the Conductor is tested with simulated events and states from the Hardware and Model. With mocks, even hardware register configuration code and ISRs can be tested. An MCH example follows in a later section of this paper.

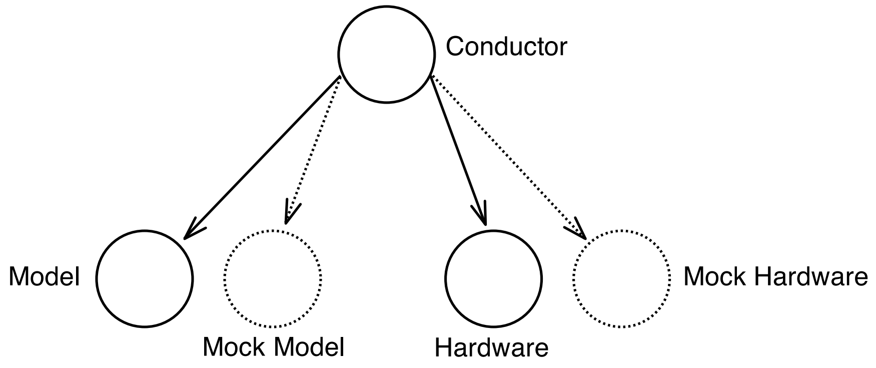

Fig. 1. Relationships of a Conductor to its complementary MCH triad members and mocks. The depicted mocks stand in for the concrete members of the triad (linked in at build time) and allow testing of the logic within the Conductor. Mocks are automatically generated capture function parameters and simulate return values used in test assertions.

Model Conductor Hardware & Multi-Triad Systems

Multiple triads are necessary to compose an entire system and simplify testing. Multiple triads exist independently

incongruity is due to the importance we place on defining the triad interactions at the Conductor first. As the Conductor is central to the behavior of an MCH triad, we elected to name the design pattern to reflect this.

of one another. A central Executor contains the system's main execution loop; it initializes and continually runs each triad by making calls on the Conductors. If there is communication necessary among triads in a multi-triad system, it will occur between Models. In a real-time system, individual triads can map well to tasks or threads.

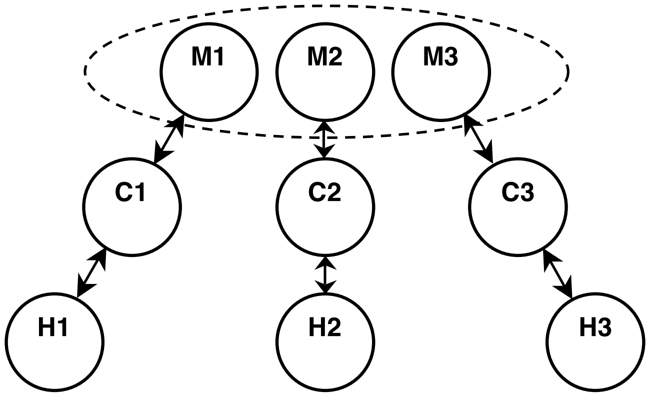

Fig. 2. A system composed of multiple MCH triads. Inter-triad communication occurs among Models through direct calls or shared helpers. An external Executor runs each triad by making calls to service each Conductor.

Overhead & Development Cost

MCH adds little to no overhead (i.e. memory use or processor time) to a production embedded system. Of course, mocks and unit tests are not compiled into the final release system. Further, MCH is essentially naming, organization, and calling conventions; any overhead incurred by these conventions is easily optimized away by the compiler.

TDD shifts the time normally spent repaying technical debt[3] at project conclusion (and beyond) to the development phase with test-first practices. With this shift comes predictable development allowing decision makers to manage project risks with progress data. With TDD and CI clear savings are realized over the system's lifetime in reduced bugs, reduced likelihood of recall, and ease of feature additions and modifications. New developers can work on the system and make changes with confidence because of test suites. Those tests, in fact, operate as "living" documentation of the system. Looking at the entire life cycle of a system, TDD and CI are considerably more cost effective than traditional or ad hoc methods of development.

Sample Project

To demonstrate the ideas in this paper, we created a full sample project for an Atmel ARM7-based development system (AT91SAM7X). This example system samples a thermistor, calculates a weighted average temperature, outputs the value through an RS232 serial port, and handles error conditions. The project includes three MCH triads: analog-to-digital conversion (ADC), timer management, and serial data output. The full project including support tools is available at http://atomicobject.com/pages/Embedded+Software.

For discussion here, we will walk through development of the ADC triad from beginning to end. Along the way we will also demonstrate the use of the tools we developed to complement the principles espoused earlier.

Summary of Steps

1. Create system tests for the temperature conversion feature to be developed.
2. Generate a skeleton Model, Conductor, Hardware, and accompanying test files.
3. Add tests and production code for initialization to the Executor, ADC triad, and helper functions.
4. Add tests and production code to the Executor, ADC triad, and helper functions for runtime temperature capture and conversion. Start with the Conductor first.
5. Verify that all unit tests and system tests pass.

Create Temperature Conversion System Tests

Using Systir, the miniLAB driver, and a serial port driver, our system test provides several input voltages to the development system (simulating a thermistor in a voltage divider circuit) and expects temperatures in degrees Celsius to be read as ASCII strings from the development PC's serial port. The system test is running externally to the development board itself.

```
proves "voltage read at temperature sensor input is translated and reported in degrees C"

# Verifies the 'stable' range of the sensor (10C-45C). Instability is due to the flattening
# of the temperature sensor voltage to temperature response curve at the extremes.

set_temperature_in_degrees_C(10.0)
verify_reported_temperature_in_degrees_C_is_about(10.0)

set_temperature_in_degrees_C(25.0)
verify_reported_temperature_in_degrees_C_is_about(25.0)

set_temperature_in_degrees_C(35.0)
verify_reported_temperature_in_degrees_C_is_about(35.0)

set_temperature_in_degrees_C(40.0)
verify_reported_temperature_in_degrees_C_is_about(40.0)
```

Fig. 3. Systir system test for verification of temperature conversion. Driver code handling timing, test voltage calculation, and communications is not shown.

Generate Skeleton MCH Triad and Unit Test Files

We create skeleton versions of the source file, header file, and test file for each member of an ADC MCH triad. The use of the generation script and an Empty ADC Conductor are shown in the following figures. For each component in the system (whether a member of a triad or a helper), this same process is repeated to create skeleton files.

```
> ruby auto/generate_src_and_test.rb AdcConductor
> ruby auto/generate_src_and_test.rb AdcModel
> ruby auto/generate_src_and_test.rb AdcHardware
```

Fig. 4. Generating a skeleton MCH triad and tests with the module generation Ruby script.

```
#ifndef _ADCCONDUCTOR_H
#define _ADCCONDUCTOR_H

#include "Types.h"

#endif // _ADCCONDUCTOR_H
```

Fig. 5. AdcConductor skeleton header.

```
#include "AdcConductor.h"
```

Fig. 6. AdcConductor skeleton source file.

```
#include "unity_verbose.h"
#include "CMock.h"
#include "AdcConductor.h"

static void setUp(void)
{
}
static void tearDown(void)
{
}

static void testNeedToImplement(void)
{
  TEST_FAIL("Implement me!");
}

//[[$argent require 'generate_unity.rb'; inject_mocks("AdcConductor");$]]
//[[$end$]]

//[[$argent require 'generate_unity.rb'; generate_unity();$]]
//[[$end$]]
```

Fig. 7. TestAdcConductor skeleton unit test file. Note inclusion of Argent blocks. Before compilation, a Rake task will execute Argent, and these blocks will be populated with C code to use the Unity test framework and CMock generated mock functions (mocks are generated from header files by another build task prior to Argent execution).

Add Tests and Production Code for Initialization

The Executor initializes the system by calling initialization functions in each triad's Conductor. The Conductors delegate necessary initialization calls. The ADC Hardware will use a helper to configure the physical analog-to-digital hardware. For brevity, the Executor's tests and implementation are not shown here.

```
static void testInitShouldCallHardwareInit(void)
{
  AdcHardware_Init_Expect();
  AdcConductor_Init();
}
```

Fig. 8. Initialization integration test within the Conductor's test file. The "_Expect" function is automatically generated by CMock from the interface specified in the Hardware header file.

```
void AdcConductor_Init(void)
{
  AdcHardware_Init();
}
```

Fig. 9. Conductor's initialization function that satisfies its unit test.

```
static void testInitShouldDelegateToConfiguratorAndTemperatureSensor(void)
{
  Adc_Reset_Expect();
  Adc_ConfigureMode_Expect();
  Adc_EnableTemperatureChannel_Expect();
  Adc_StartTemperatureSensorConversion_Expect();

  AdcHardware_Init();
}
```

Fig. 10. Initialization integration test within the TestAdcHardware's test file. Calls are made to an AdcHardwareConfigurator helper and an AdcTemperatureSensor helper. The "_Expect" and "_Return" functions are automatically generated by CMock from the interfaces specified in header files.

```
void AdcHardware_Init(void)
{
  Adc_Reset();
  Adc_ConfigureMode();
  Adc_EnableTemperatureChannel();
  Adc_StartTemperatureSensorConversion();
}
```

Fig. 11. AdcHardware's initialization function that satisfies its integration test.

```
static void testResetShouldResetTheAdcConverterPeripheral(void)
{
  AT91C_BASE_ADC->ADC_CR = 0;
  Adc_Reset();
  TEST_ASSERT_EQUAL(AT91C_ADC_SWRST, AT91C_BASE_ADC->ADC_CR);
}

static void testConfigureModeShouldSetAdcModeRegisterAppropriately(void)
{
  uint32 prescaler = (MASTER_CLOCK / (2 * 2000000)) - 1; // 5MHz ADC clock
  AT91C_BASE_ADC->ADC_MR = 0;
  Adc_ConfigureMode();
  TEST_ASSERT_EQUAL(prescaler, (AT91C_BASE_ADC->ADC_MR & AT91C_ADC_PRESCAL) >> 8);
}

static void testEnableTemperatureChannelShouldEnableTheAppropriateAdcInput(void)
{
  AT91C_BASE_ADC->ADC_CHER = 0;
  Adc_EnableTemperatureChannel();
  TEST_ASSERT_EQUAL(0x1 << 4, AT91C_BASE_ADC->ADC_CHER);
}
```

Fig. 12. AdcHardwareConfigurator's initialization unit tests.

```
void Adc_Reset(void)
{
  AT91C_BASE_ADC->ADC_CR = AT91C_ADC_SWRST;
}

void Adc_ConfigureMode(void)
{
  AT91C_BASE_ADC->ADC_MR = (((uint32)11) << 8) | (((uint32)4) << 16);
}

void Adc_EnableTemperatureChannel(void)
{
  AT91C_BASE_ADC->ADC_CHER = 0x10;
}
```

Fig. 13. AdcHardwareConfigurator's initialization functions that satisfy its unit tests.

```
static void testShouldStartTemperatureSensorConversionWhenTriggered(void)
{
  AT91C_BASE_ADC->ADC_CR = 0;
  Adc_StartTemperatureSensorConversion();
  TEST_ASSERT_EQUAL(AT91C_ADC_START, AT91C_BASE_ADC->ADC_CR);
}
```

Fig. 14. AdcTemperatureSensors's start conversion unit test.

```
void Adc_StartTemperatureSensorConversion(void)
{
  AT91C_BASE_ADC->ADC_CR = AT91C_ADC_START;
}
```

Fig. 15. AdcTemperatureSensors' start conversion function that satisfies its unit test.

Add Tests and Production Code for Temperature Conversion

The models of all the triads are tied together for inter-triad communication. The timing Model repeatedly updates a task scheduler helper with time increments. The ADC Model returns to the Conductor a boolean value processed by the task scheduler helper to control how often an analog-to-digital conversion occurs.

The ADC Conductor is repeatedly serviced by the Executor once initialization is complete (the other triad Conductors are serviced in an identical fashion). Each time the ADC Conductor is serviced, it checks the state of the ADC Model to determine whether an analog-to-digital conversion should occur. When a conversion is ready to occur, the ADC Conductor then instructs the ADC Hardware to initiate an analog-to-digital conversion. Upon completion of a conversion, the Conductor provides the raw value in millivolts to the ADC Model for temperature conversion. The timing and serial communication triads will cooperate to average and periodically output a properly formatted temperature string. Here, we walk through the tests and implementation of the ADC Conductor's interaction with the ADC Hardware and ADC Model down through to the hardware read of the analog-to-digital channel.

```
static void testRunShouldNotDoAnythingIfItIsNotTime(void)
{
  AdcModel_DoGetSample_Return(FALSE);
  AdcConductor_Run();
}

static void testRunShouldNotPassAdcResultToModelIfSampleIsNotComplete(void)
{
  AdcModel_DoGetSample_Return(TRUE);
  AdcHardware_GetSampleComplete_Return(FALSE);
  AdcConductor_Run();
}

static void
testRunShouldGetLatestSampleFromAdcAndPassItToModelAndStartNewConversionWhenItIsTime(void)
{
  AdcModel_DoGetSample_Return(TRUE);
  AdcHardware_GetSampleComplete_Return(TRUE);
  AdcHardware_GetSample_Return(293U);
  AdcModel_ProcessInput_Expect(293U);
  AdcHardware_StartConversion_Expect();
  AdcConductor_Run();
}
```

Fig. 16. TestAdcConductor integration tests for interactions with AdcHardware and AdcMcodel. The "_Expect" and "_Return" functions are automatically generated by CMock from the interfaces specified in header files.

```
void AdcConductor_Run(void)
{
  if (AdcModel_DoGetSample() && AdcHardware_GetSampleComplete())
  {
    AdcModel_ProcessInput(AdcHardware_GetSample());
    AdcHardware_StartConversion();
  }
}
```

Fig. 17. AdcConductor's run function (called by the Executor) that satisfies the Conductor unit tests.

```
static void
testGetSampleCompleteShouldReturn_FALSE_WhenTemperatureSensorSampleReadyReturns_FALSE(void
)
{
  Adc_TemperatureSensorSampleReady_Return(FALSE);
  TEST_ASSERT(!AdcHardware_GetSampleComplete());
}

static void
testGetSampleCompleteShouldReturn_TRUE_WhenTemperatureSensorSampleReadyReturns_TRUE(void)
{
  Adc_TemperatureSensorSampleReady_Return(TRUE);
  TEST_ASSERT(AdcHardware_GetSampleComplete());
}

static void testGetSampleShouldDelegateToAdcTemperatureSensor(void)
{
  uint16 sample;
  Adc_ReadTemperatureSensor_Return(847);

  sample = AdcHardware_GetSample();
  TEST_ASSERT_EQUAL(847, sample);
}
```

Fig. 18. Integration tests for AdcHardware. Note that the "_Expect" and "_Return" functions are automatically generated by CMock from the interfaces specified in header files.

```
void AdcHardware_StartConversion(void)
{
  Adc_StartTemperatureSensorConversion();
}

bool AdcHardware_GetSampleComplete(void)
{
  return Adc_TemperatureSensorSampleReady();
}

uint16 AdcHardware_GetSample(void)
{
  return Adc_ReadTemperatureSensor();
}
```

Fig. 19. AdcConductor's functions satisfying the Conductor integration tests. Note that AdcHardware calls helper functions in AdcTemperatureSensor

```
static void testShouldStartTemperatureSensorConversionWhenTriggered(void)
{
  AT91C_BASE_ADC->ADC_CR = 0;
  Adc_StartTemperatureSensorConversion();
  TEST_ASSERT_EQUAL(AT91C_ADC_START, AT91C_BASE_ADC->ADC_CR);
}

static void testTemperatureSensorSampleReadyShouldReturnChannelConversionCompletionStatus(void)
{
  AT91C_BASE_ADC->ADC_SR = 0;
  TEST_ASSERT_EQUAL(FALSE, Adc_TemperatureSensorSampleReady());
  AT91C_BASE_ADC->ADC_SR = ~AT91C_ADC_EOC4;
  TEST_ASSERT_EQUAL(FALSE, Adc_TemperatureSensorSampleReady());
  AT91C_BASE_ADC->ADC_SR = AT91C_ADC_EOC4;
  TEST_ASSERT_EQUAL(TRUE, Adc_TemperatureSensorSampleReady());
  AT91C_BASE_ADC->ADC_SR = 0xffffffff;
  TEST_ASSERT_EQUAL(TRUE, Adc_TemperatureSensorSampleReady());
}

static void testReadTemperatureSensorShouldFetchAndTranslateLatestReadingToMillivolts(void)
{
  uint16 result;

  // ADC bit weight at 10-bit resolution with 3.0V reference = 2.9296875 mV/LSB
  AT91C_BASE_ADC->ADC_CDR4 = 138;
  result = Adc_ReadTemperatureSensor();
  TEST_ASSERT_EQUAL(404, result);

  AT91C_BASE_ADC->ADC_CDR4 = 854;
  result = Adc_ReadTemperatureSensor();
  TEST_ASSERT_EQUAL(2502, result);
}
```

Fig. 20. Unit tests for AdcTemperatureSensor helper.

```
void Adc_StartTemperatureSensorConversion(void)
{
  AT91C_BASE_ADC->ADC_CR = AT91C_ADC_START;
}

bool Adc_TemperatureSensorSampleReady(void)
{
  return ((AT91C_BASE_ADC->ADC_SR & AT91C_ADC_EOC4) == AT91C_ADC_EOC4);
}

uint16 Adc_ReadTemperatureSensor(void)
{
  uint32 picovolts = ConvertAdcCountsToPicovolts(AT91C_BASE_ADC->ADC_CDR4);
  return ConvertPicovoltsToMillivolts(picovolts);
}


static inline uint32 ConvertAdcCountsToPicovolts(uint32 counts)
{
  // ADC bit weight at 10-bit resolution with 3.0V reference = 2.9296875 mV/LSB
  uint32 picovoltsPerAdcCount = 2929688;

  return counts * picovoltsPerAdcCount; // Shift decimal to preserve accuracy in fixed-point
}

static inline uint16 ConvertPicovoltsToMillivolts(uint32 picovolts)
{
  const uint32 halfMillivoltInPicovolts = 500000;
  const uint32 picovoltsPerMillivolt = 1000000;

  // Add 0.5 mV to result so that truncation yields properly rounded result
  picovolts += halfMillivoltInPicovolts;

  return (uint16)(picovolts / picovoltsPerMillivolt); // Divide to convert to millivolts
}
```

Fig. 21. AdcTemperatureSensor helper functions satisfying the AdcTemperatureSensor unit tests.

Verify that all unit tests and system tests pass.

The process of adding tests and production code to satisfy a system requirement, of course, requires an iterative approach. Design decisions and writing test and production code require multiple passes. The code samples presented in the preceding section represent finished product. As the sample project was developed, tests and code were refactored and run numerous times. With the automation provided by our unit and system test frameworks, CMock, and the build tools, this process was nearly painless.

```
> rake clean
> rake test:units
> rake test:system
```

Fig. 22. Running rake tasks to build and test the system. The definitions of the rake tasks and the calls to CMock & Argent, dependency generation, compiling, and linking are defined within the Rakefile and not shown.

Conclusion

Picture a desperate embedded systems engineer surrounded by wires, boards, and scopes, trying to track down a mysterious bug in a product that's three weeks past its deadline. Now imagine a calm, productive developer over the course of many months regularly defining system behavior through tests and checking in code for the continuous integration server to build and test. Further, imagine a project manager confidently estimating and adjusting project completion goals and costs from measurable progress metrics. Regression testing, decoupled design, and disciplined development can all but eliminate heroic debugging efforts and costly defects released into production.

Acknowledgments

Authors wish to thank Matt Werner of Savant Automation for the opportunity to implement and originally prove out in a production system the ideas that inspired this paper. Authors also wish to thank their colleagues at Atomic

Object for their many contributions to this work including but not limited to Presenter First, Systir, the dependency generation tool, and, of course, much proofreading and editing assistance.

References

[1] Kent Beck. Extreme Programming Explained. Reading, MA: Addison Wesley, 2000.

[2] Michael Karlesky, William Bereza, and Carl Erickson. “Effective Test Driven Development for Embedded Software.” IEEE EIT2006. East Lansing, Michigan. May 2006.

[3] Technical Debt, http://www.c2.com/cgi/wiki?TechnicalDebt

[4] Nancy Van Schooenderwoert. “Embedded Agile: A Case Study in Numbers”, http://www.ddj.com/dept/architect/193501924, November 6, 2006.

[5] Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes. “Mock Roles, not Objects.” OOPSLA 2004. Vancouver, British Columbia.

[6] M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, C. Stienstra. “Presenter First: Organizing Complex GUI Applications for Test-Driven Development.” Agile 2006. Minneapolis, MN. July 2006.

[7] System Testing in Ruby, http://atomicobject.com/pages/System+Testing+in+Ruby

[8] Ruby Programming Language, http://www.ruby-lang.org/en/

[9] Domain Specific Language, http://en.wikipedia.org/wiki/Domain_Specific_Language

[10] Measurement Computing Corp. miniLAB 1008, http://www.measurementcomputing.com/cbicatalog/cbiproduct_new.asp?dept_id=412&pf_id=1522

[11] Argent, http://rubyforge.org/projects/argent/

[12] Kent Beck. “Simple Smalltalk Testing: With Patterns,” http://www.xprogramming.com/testfram.htm.

[13] http://atomicobject.com/pages/Embedded+Software

[14] Martin Fowler. “Dependency Injection,” http://www.martinfowler.com/articles/injection.html.

[15] http://rake.rubyforge.org/

[16] David Crosby and Carl Erickson. “Big, Complex, and Tested? Just Say ‘When’: Software Development Using Presenter First”, Better Software Magazine. February 2007.

Appendix

Test-Driven Development Overview

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. In TDD, development is not writing all the functional code and then later testing it, nor is it verifying code by stepping through it with a debugger. Instead, testing drives development. A developer looks for ways to make the system testable, does a small amount of design, writes test programming for the piece of the system currently under development, and then writes functional code to meet the requirements of the test-spawned design. Designing for testability in TDD is a higher calling than designing “good” code because testable code *is* good code.

At the highest levels (e.g. integration and system testing) fully automated testing is unusual. However, at the lowest level, automated unit testing is quite possible. 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) 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 high-level integration or system testing will complement these unit tests and ideally will include some measure of automation.

System Test-Driven Development follows these steps:

1. Pick a system feature.
2. Program a system test to verify that feature.
3. Compile; run the system test with the system itself and see it fail.
    4. Identify a piece of functionality within the feature (a single function or method).
    5. Program integration and unit tests to verify that functionality.
    6. Stub out the functional code under test (to allow the test code to compile).
    7. Compile; run the integration and unit tests and see them fail (to verify expectations).
    8. Flesh out the functional, production code.
        9. Compile; run the integration and unit tests.
        10. Refactor the production code.
        11. Repeat 9-10 until the integration and unit tests pass and the functional code is cleanly implemented.
    12. Compile; run the system test.
    13. Repeat 4-12 until the system test passes.
14. Repeat 1-13 until all features of the system are implemented.

TDD provides several clear benefits:

- Code is always tested.
- Testing drives the design of the code. As a side effect, the code is well designed because of the decoupling necessary to create testable code.
- The system grows organically as more knowledge of the system is gained.
- The knowledge of the system is captured in tests; the tests are “living” documentation.
- Developers can add new features or alter existing code with confidence that automated regression testing will reveal failures and unexpected results an d interactions.
- Tests catch the majority of bugs and leave for a human mind difficult testing issues like timing collisions or unexpected sub-system interactions.

Continuous Integration Overview

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 complements to effective TDD. When TDD and CI are used together the system’s code-base is always thoroughly tested and has few, if any, integration problems among subsystems or sections of code. Integration problems are discovered early when it is cheapest to correct them. Further, any such problem will be discovered close to where and when the problem was created; here, understanding is greatest and good design choices are most likely.