A look at test driven development (TDD) in the embedded environment.

Examples of tests run of the ADSP-BF533 hardware and software

M. R. Smith, J. Miller, A. Geras et al.
Department of Electrical and Computer Engineering
University of Calgary
Alberta, Canada

Email: smithmr @ ucalgary.ca
Phone: 1-403-220-6142

This article is NOT for general distribution at this point in time

This article has been accepted by Circuit Cellar Magazine (November 2004) publication in special Embedded System Issues March 2005 and April 2005

Copyright M. R. Smith – November 2004
Thermal Arm Wrestling
A look at test driven development (TDD) in the embedded environment.

Is Test Driven Development (TDD) in the embedded environment an important new idea, or simply just enough formal training to provide an old-dog a better way to do what was done before?

Mike Smith, Alan Martin, Lily Huang, Moreno Bariffi, Andrew Kwan, Warren Flaman, Adam Geras and James Miller

One of the new buzz concepts coming out of the software engineering society is agile software development. In general terms, agile processes are styles of specifying, designing, building and testing software that are very collaboration intensive. To guarantee a high level of collaboration, agile processes frequently involve team-based activities and pair-based activities in the normal course of a working day. Agile proponents strongly favour face-to-face communication over swapping a large number of documents amongst team members.

One key Agile element is the concept of Test Driven Development (TDD) where you use tests up front to specify the target requirements and to describe a target design (Fig. 1A). This contrasts to the more traditional Test Last Development (TLD) process where requirements gathering, high and low design documents form a major, upfront time commitment (Fig. 1B) before the majority of testing issues are really considered.

Test driven development is not just letting people “get on with the coding”. The requirements are still analyzed, but the analysis is performed in close consultation with the customer, with the customer providing a series of tests. The developer then writes “the simplest possible code” that will satisfy the customer tests. Since an important expectation with test driven development is that the tests be automated and straight-forward to run, the developer is encouraged to use them frequently during the project’s various development stages. Prior to developing any code, the developer generates additional tests which, with the code working, will be used in the style of post-development “unit tests” from a traditional manner software process.

With the product now capable of producing the required result, meeting the customer requirements, code development moves in a new direction. The developer now changes or “re-factors” the code; making modifications for reliability, rather than functionality. In this phase of the project, the mode for using the “tests developed up-front” has switched to something that looks more conventional; post-production testing.

Up-front test development sounds, in principle, a good idea especially since all important customer and developer tests are not left to the time allotment vagaries and pressures at the end of the project. However research reports and anecdotal comments are split about which of TDD and TLD is really the most efficient (highest developer productivity), provides the best quality code (reduced number of post-release problems) or best handles the many other customer and developer related issues that arise during code production and delivery.

Mike (the old-dog referred to on the first page) has been developing computer programs for embedded systems since the earlier 70’s, and can be considered as a strict TLD traditionalist. He does actually write tests for his code, but they often have been more of the “debug” rather any formal testing format.

```c
#define USE_TESTS

value = Process_Array(array1, array2);

#if defined(USE_TESTS)

#endif
```
The other tests are of the practical, non formal “provides the functionality I expect – what more is needed?” type of test from a small “one-off” project amateur.

Adam and James, on the other hand, are software project developers with considerable industrial experience using testing environments such as the popular xUNIT frameworks (JUnit, NUnit, CppUnit, etc). However, such big “desk-top” systems have “mega-mega bytes” for running a testing environment and are not limited by the memory and real-time constraints found in an embedded systems environment.

The questions that needed to be answered were:
- Can you develop a TDD tool suitable for the constrained embedded system environment?
- Could a small project amateur gain anything from using such a more formalized testing approach?

Note: Some of the chronology in the development of this project and its various components have been changed to make the story more interesting.

Project concept – customer requirements

A high school engineering recruitment project was chosen as the target for this investigation. This project had to be quick and reliable to set-up, a little flashy, not over-powering, and just corny enough to create interest in engineering amongst students between grades 10 and 12.

“Thermal arm wrestling” seemed to fit the target. Two pulse-width modulated thermal sensors (Analog Devices TMP03) are “warmed” by the student’s thumbs (Fig. 2a). The sensor signals are fed to an embedded system and converted into temperature values. The difference in temperatures between the two competitors is used to control a cursor on a TV game screen (Fig. 2b).

To introduce an “engineering design element” into the game, two game strategies are provided for each competitor. You can thermally overpower your opponent and move the cursor into the win area furthest from you. Alternately, you can “psyche out” your opponent by having the sensor cool after the start of the game, causing the cursor to overshoot their “hot winning area” into your “cold winning area”.

Test Driven Development environment

Prior to developing the game, we had been modifying a “C++” automated testing environment to run within the available memory space on an embedded system as part of a University-industrial collaborative research and development project. We chose to adapt the small non-scripted CppUnitLite (http://c2.com/cgi/wiki?CppUnitLite) developed by Michael Feathers. “Non-scripted” means that the framework detects the tests to be run automatically, freeing the developer from having to write a script that specifically runs the appropriate tests.

A key element in adapting this tool for the embedded system was to minimize memory usage to ensure that it would fit within the embedded system environment. In particular, any print statements which involved formatting were replaced by simpler statements (puts(astring)). The sheer generality of the formatting associated with statements such as cout << value or printf(“%d”, value) can generate sufficient instructions to occupy most of the available program memory space; possibly leaving room for little other code.

However, these changes don’t solve all the problems associated with the reports generated during test driven development on a “live” rather than “simulated” environment. Messages are sent back from the target to the development environment over a serial, USB or JTAG connection. Transmitting strings over such a connection is a complex task. Basically the target must be stopped and switched into “emulator interrupt” mode. This permits the message to be “wangled” out of the target “one character at time”, but completely disrupts the real-time operation of the embedded systems. For some of the development environments we have
Test driven development for embedded systems examined, there is a “background telemetry channel” specifically designed to permit message interchange with no, or minimal, disruption of real time operation.

As will be demonstrated later in the article, we added some specialized embedded system specific capabilities. These included the ability to watch (in real time) the performance of specific processor peripheral registers during system initialization and time critical sections of code. Additional tests had to be added to overcome an unexpected problem when testing unsigned short variables. Such variables are extensively used when programming devices with application registers of various different byte sizes.

For more information on the current state of our test driven development environment research visit http://www.enel.ucalgary.ca/People/Smith/embeddedTDD/

A neophyte develops a Customer requirement’s test.

In a “real” test driven development environment, we would immediately talk to the customer (ourselves) and code up some “customer tests” for our code to satisfy. However, since we are neophytes at this TDD game the best customer requirement tests we could develop are shown in Listing 1.

As can be see, main() calls run_tests(); a key element within the TDD environment. Figuratively, although not literally, run_tests() sets up a list of available tests (CUSTOMER, UNIT, DEVELOPER, EXPERT etc.) and runs them. TEST() and CHECK() are examples of the available test macros. The macro format is used as it permits descriptive strings to be automatically constructed during the program compile to provide information about which successes and failures are being reported. Automation and ease of use are key elements if the developer is actually going to use the testing environment.

Note that there will be 4 failures when this test is run. This is an expected feature of TDD at this point in the hardware-software co-design process. The 4 failures occur due to the fact that there is currently no code or hardware available to satisfy the test. After all this is a test driven development where the tests are written before the code; rather than after the code as with test last development. Note the single pass. This is essentially an indication that the test ran – simple positive re-inforcement.

To put at least some sort of positive spin on the customer test shown in Listing 1, we can say that the test is automated; and one item is to be tested in two different ways; that must count associated with the fact that the test series did actually run. Note: the test IS at this point in time, no code (other than function stubs) exist. The single pass is customer requirements in the form of test. There are 4 failures because at this point in time, no code (other than function stubs) exist. The single pass is associated with the fact that the test series did actually run. Note: the test IS automated and one item is to be tested in two different ways; that must count something for!

lets investigate the advantages (or otherwise) of TDD in a context with which we are more familiar – developing the code.

Making the typical developer’s best guesstimate of what the customer “really wants”, we can proceed with the overall system requirements and high level design. In describing the target architecture, we will again use tests to guide our work. Tests will be used to describe the intended behavior of the target components as they interact to satisfy the perceived customer requirements. Since these are lower-level tests and the customer may not be aware of the need for them, these tests are referred to as “developer tests” – primarily based on the fact that the developer identifies the tests cases and the expected results of each of them. Exploiting the synergy between customer tests and developer tests is another of those ‘hard to learn’ skills in effective test-driven development. Given that our problem is small in this example, it is unlikely that the relationship between customer and developer tests will have too big an impact on the way we need to proceed.

Copyright M. R. Smith – November 2004
Game paddles

The output from the ADSP-TMP03 thermo-sensors is basically a 33 Hz logic-level (5V) square wave whose duty cycle varies with temperature (Fig. 3). Temperature can be calculated using the formula

\[
\text{Temperature} = A - B \times \frac{\text{TIME\_HIGH}}{\text{TIME\_LOW}}
\]

A key feature of the device is that it depends on the ratio of the times for the high and low signal cycles. This means that repeatability of measured times is more important than the absolute accuracy of those times. The constants \(A\) and \(B\) depend on whether you want the output in Celsius or Fahrenheit. (Let’s think ahead, we may want to market this game in both Canadian and US markets!)

Since we are dealing with simple logic levels (0V, 5V), only a simple digital signal input needs to be analyzed rather than an analog signal or a parallel interface.

**Video requirements**

Video is obviously going to be a key element of making an attractive display for prompting engineering at high schools. The long term goal is to develop video images “on-the-fly” and output images that change in response to information from the game paddles.

However, we currently don’t know how to handle simultaneous generation and display of images; and the first required demo is for Friday; and today is Monday! What we will do is to take a simpler initial approach and store 30 predefined images in the evaluation board’s SRAM. Each image will have the same background, but with the cursor placed in slightly different positions. We can use DMA controlled interrupts to transfer different images to the TV screen on demand. This is essentially a multiple buffer approach to handling the game screen. It’s also the ‘simplest possible solution’—indicative of using another one of the agile development practices that encourages developers to show progress early by using the simplest design that could possibly work.

**Development system**

Just recently one of the authors (MS) had decided to take the challenge of upgrading a 3rd year “hands-on” undergraduate course from some “well-used” and “well understood” MC68332 evaluation boards to something more modern. Given the need for long-term flexibility in available laboratory tasks (to avoid road-mapping issues) an evaluation board with many capabilities was needed.

The Analog Devices ADSP-BF533 (Blackfin) EZ-Lite evaluation board (Fig. 4) was chosen for this course (http://www.enel.ucalgary.ca/People/Smith/2004webs/encm415_04/). Both the flexibility and availability of these boards made them an ideal target for the “thermal-arm wrestling” project.

- Available SRAM for storage of images
- Available FLASH memory for ease of updates, and automatic boot-up without requiring an external control computer
- Available programmable flags (PF) for digital input.
- Available audio and video signals. We had already decided that version 2 of the game would involve using phase shifts to control the apparent position of a sound source in response to temperature changes from the game paddles. Nintendo – here we come!
- Lots of processor speed, so we could do most of the prototyping in optimized “C++” and anticipate not having to worry about timing issues (moving code into parallel assembly). This is important as it is now late Tuesday morning, and the first demo is still on Friday.

![Figure 4. The 500 MHZ Analog Devices ADSP-BF533 (Blackfin) EZ-Lite evaluation board has audio and video capability, digital inputs. Most important, there are six LEDs that can be flashed – apparently a key-element for the customer who wants something to be happening when the high school student looks at the processor board!](http://www.enel.ucalgary.ca/People/Smith/2004webs/encm415_04/)

---

**Figure 3.** Examples of the timing diagram for the ADSP-TMP03 thermo-sensor output for a lower (a) and higher (b) temperatures. As the temperature of the thermo-sensor increases, the duty cycle of the square wave changes; although not as dramatically as illustrated in the figure.
The modified test-driven development environment source code was compiled, linked and downloaded to the embedded system with no changes to the standard environment running on the VisualDSP++ IDE provided with the evaluation board. Given that the code was written in "C++", we were not expecting any compatibility issues. However, we were concerned about code size issues. The TDD environment passed this first test; we are now capable of running the first test (Listing 1) and can truthfully expect all the tests to pass. We know that no code was written, but when the Blackfin board boots up it goes in self test mode and the LEDs blink, meeting a key customer requirement. However, shipping at this point is unlikely to generate any further orders!

The original TDD environment requires a number of key extensions for use in the embedded systems environment. First there is a requirement to be able to reconfigure the hardware environment to a known state prior to issuing a series of tests. To capture this functionality, we developed three new hardware oriented TDD procedures

1. **__CaptureKnownState( )** – This procedure is called as the first line of a main() function run on a board that has just been powered up. It automatically saves the “C++” initial environmental setup to a file. The function was written to be easily upgraded as new features of the embedded system are added (new peripherals developed etc).

2. **__SaveUserRegAndReset( )** and **__RecoverUserReg( )** are a pair of functions to be used at the start and end of a test respectively. The first saves the current user processor state and resets the system to a known state, whilst the second restores the initial user processor state.

The jury is out on the utility of these functions. It is true that having the system in a known state before testing prevents many errors. However, we have found that testing with the system that is unintentionally in an unknown state does also uncover unexpected system configuration issues.

A second useful hardware test utility is the ability to watch planned (or unintentional) action in a memory range on the processor.

**WatchDataClass** and **WATCH_MEMORY_RANGE( )**

This test is performed using “hardware breakpoints” on a running system, rather than via “static profiling” on an architectural simulator or “statistical profiling” (snapping a quick look) on a running system. To be honest, this test class was initially developed as an INSTRUCTOR test. Essentially an INSTRUCTOR (with very intimate prior knowledge of the system architecture) might write such a test to automatically examine whether students learning about a processor have properly configured the registers of a peripheral correctly.

However, in practice, this test class proved to be much more utilitarian. This is a test that might be useful when the developer knows exactly what needs to be done, and how to do it, but gets distracted by an interruption in the middle of the creative process. When used this way, it is probably better called the AVOID_LSD_ERROR test class where LSD stands for those Little Stupid Details which cause such a great waste of time amongst experienced developers. The full advantages of such a test are discussed later.

The final **TimeClass** is a series of timing utilities and test. The first permits a measurement of the execution time of a function **MEASURE_EXECUTION_TIME(Func(pars))**. The second is an actual test **MAXTIME_ASSERT(Func(pars))** to determine whether a function satisfies some strict real-time test. This is the first of a series of functional and non-functional tests for embedded systems planned for development. A examples of a non-functional test is **MAXPOWER_ASSERT( )** used to determine, as a design requirement, whether a given algorithm meets the specific power restriction designated by the customer.

The timer and watch-data classes are general in concept, but must be specifically implemented to meet the available resources on a given processor; but, at the same time, not restrict the normal development of code. On the Blackfin processor there are specific hardware Watch registers that can determine the number of reads and /or writes in either a data or program RAM. These were used for the watch data class. The timer class was developed using the system clock which can be expected to be either available for use, or operating in a predictable fashion.

Details of the current TDD environment tested on Analog Devices integer Blackfin (ADSP-BF5XX) and VLIW TigerSHARC (ADSP-TSXXX) processors can be found at <http://www.enel.ucalgary.ca/People/Smith/embeddedTDD/>.

**Software-hardware co-design using a test driven development frame of mind.**

Although it may be difficult for an “old-dog” to develop appropriate customer tests “ahead of time”, that is not the case for developer tests. The developer already has in mind a specific expected result before the code has been developed. Whether a test for this result is developed before or after the code is developed is just a mind set.

Key elements in test driven development are “writing the simplest code to satisfy the customer test” and “refactoring the code to improve reliability rather than functionality”. Again, being honest, the first element makes sense, but I can only go along with the second element just so far. Refactoring sounds good, but as a dedicated amateur I rather tend to adhere to the principle “if it ain’t broke -- don’t fix it”. However it would be nice to have a test around and easily to use just in case I did choose to modify, rather than fix, existing code. After all program modifications required to accommodate new or extended features suggested during customer consultations are probable, rather than just likely. In addition, since tests can be created and run so easily, developers will be encouraged to use more tests in the battle to develop software that is working, on time and on budget. I have my suspicions as I have read in books how they used to “encourage the others” during a different battle – the trenches of...
the First World War! One further advantage of having proper customer tests available is that unrecognized scope creep is less likely to happen.

Most of the code for generating and displaying the signals using the video codec on the evaluation board is a modification of provided examples. Remember it is now Wednesday afternoon and the product must be demonstrated to our customer’s customers on Friday morning. This means that we make some executive decisions. Next week, we will generate a series of acceptance tests to see how reliable this reused code is. Right now, if something is displayed on the screen, and the picture does not flicker, then that’s great!

The code necessary to control the game paddles revolves around more critical software and hardware components. These are not standard features and must be custom constructed.

Tests for temperature calculation.

Using the ADSP-TMP03 data sheet and a calculator we can design a series of tests for the major software component

CalculateTemperature(highTime, lowTime, mode)

Here mode is CELSIUS or FAHRENHEIT and the highTime and lowTime are the periods obtained by measuring the transitions in the temperature sensor signals (See Fig. 3). The tests are shown in Listing 2.

Both tests to measure expected successes and possible error modes must be generated. Note that certain design decisions are specifically expressed within the test code. These decisions include the simple and the more significant. The simple decisions involve the definitions of constants to be used. The more complex is the temperatureValue4 definition indicating that temperatures will be stored as block-floating point integers temperatures returned accurate to the nearest ¼ of a degree (21.5 C returns as the short int value 82).

The top part of listing 3 shows the obvious way to write the code to “pass the tests”; doing the calculation in floating point. The software floating point emulation mode will be slow on an integer processor, but the Blackfin has a high system clock frequency and we only have a simple task with limited calculation. In actual fact, this particular code does not pass all the tests because of the way that it has been written. Since the conversion from floating point to integer occurs prior to multiplication by 4 (to give the intended ¼ degree accuracy) then temperature values such as 20.5 are incorrectly calculated. This information was revealed as soon as the predefined tests were run.

The lower part of listing 3 shows the refactored code taking into account the fact that floating point emulation on the integer Blackfin processor is slow. Note – this code passes the tests, but this does not necessary indicate that the code is correct; perhaps other tests are required.

```c
#define CELSIUS 1
#define FAHRENHEIT 2
#define GARBAGE 3

typedef temperatureValue4 signed short;

TEST(CELSIUS_20, CalculateTemperature) {
    // Time high / low values for 20 C
    temperatureValue4 value;
    value = CalculateTemperature(400, 215, CELSIUS);
    SHORTS_EQUAL(value, 80);
}

TEST(CELSIUS_20_5, CalculateTemperature) { // 20.5 C
    temperatureValue4 value;
    value = CalculateTemperature(4000, 2145, CELSIUS);
    SHORTS_EQUAL(value, 82);
}

TEST(FAHRENHEIT_NEG20, CalculateTemperature) {
    // Time values for -20 F – special Canadian test
    temperatureValue4 value;
    value = CalculateTemperature(720, 475, FAHRENHEIT);
    SHORTS_EQUAL(value, 0xFFb0);
}

TEST(GARBAGE_ERROR, CalculateTemperature) {
    temperatureValue4 value;
    value = CalculateTemperature(7200, 4745, GARBAGE);
    SHORTS_EQUAL(value, TEMPERATURE_ERROR_CODE);
}

TEST(tempZERO, CalculateTemperature) {
    temperatureValue4 value;
    value = CalculateTemperature(0, 0, CELSIUS);
    SHORTS_EQUAL(value, TEMPERATURE_ERROR_CODE);
}

Listing 2: The tests developed for CalculateTemperature() function. Note that both tests for possible failure modes that might occur are generated in addition to expected successful tests. Note that the tests also indicate certain basic design decisions such as the format of temperature values – integers with values that represent the temperature to the nearest ¼ of a degree.
```
Tests for hardware set-up

The major hardware component involves the need to simultaneously read the asynchronous signals from the two temperature sensors, together with determining the values of two switches used to reset and start the game. There are 4 inputs – programmable flags (PF) – easily accessible on the EZ-kit evaluation board.

The original concept was to use an approach where changes on the PF lines caused interrupts. The switch-driven interrupts caused changes in various flags, allowing the main task to transition between “waiting for the game to start” state, the “game-playing” state and “resetting the game while the game is running or if a winner has been determined” state. The interrupts on the temperature sensor driven lines would be used to access the value of a free-running core timer. This timing information could then be processed to generate the temperature values.

However, using the TDD approach of “easiest way to satisfy the customer requirements”, we decided instead to use a single core

timer driven interrupt rather than 4 PF-driven interrupts. During the timer interrupt routine, we could determine the sense of the input signals and therefore set the necessary flags to transfer between the main task states. We could also simply count the number of interrupts that occurred between the various high-to-low and low-to-high transitions of the temperature sensors. These counts are sufficient as only a ratio of times, rather than absolute times, as is needed to calculate the temperature. Sufficient precision in the time counts could be guaranteed by having the timer interrupts occur frequently enough.

Generating the timer tests.

Setting up the Blackfin core timer should be a simply enough task, you would think. You need to place the required period (reload value) into the TPERIOD register and the initial count into the TCOUNT register. A fourth constant value is placed into another register. However, a different test developed by the “students” reveals a fatal flaw in the code developed by the “master”.

Listing 3. The upper code, written the “simplest, obvious way”, does not satisfy all the tests as the cast from floating point occurs at the wrong time and introduces a loss in the required precision. The lower code is a refactored version that provides the required precision and is more appropriate for the integer Blackfin processor.

```
# define SCALE 0
# define PERIOD 0x2000
# define COUNT 0x2000

typedef ulong unsigned long int;

TEST(Test_SetCoreTimer, INSTRUCTOR) {
   // SaveUserRegAndReset () ;
   WatchDataClass<unsigned long> coretimer_reg(4, pTCNTL, pTPERIOD, pTSCALE, pTCOUNT);

   // Setup expected values
   unsigned long expected_value[] = {
      0x0, PERIOD, SCALE, COUNT};

   WATCH_MEMORY_RANGE(coretimer_reg, (SetCoreTimerASM(COUNT, PERIOD, SCALE)),
      READ_CHECK | WRITE_CHECK);
   __SaveUserRegAndReset () ;
   CHECK(coretimer_reg.getReadsWrites() = = 4);
   ARRAYS_EQUAL(expected_value, coretimer_reg.getFinalValue(), 4);
}
```

Listing 4: The test for the SetupCoreTimer() is straightforward. 3 known parameters are passed into the function for direct placement into the core timer register. A fourth constant value is placed into another register. However, a different test developed by the “students” reveals a fatal flaw in the code developed by the “master”. 

```
typedef ulong unsigned long int;

TEST(Test_SetCoreTimer, INSTRUCTOR) {
   // SaveUserRegAndReset () ;
   WatchDataClass<unsigned long> coretimer_reg(4, pTCNTL, pTPERIOD, pTSCALE, pTCOUNT);

   // Setup expected values
   unsigned long expected_value[] = {
      0x0, PERIOD, SCALE, COUNT};

   WATCH_MEMORY_RANGE(coretimer_reg, (SetCoreTimerASM(COUNT, PERIOD, SCALE)),
      READ_CHECK | WRITE_CHECK);
   __SaveUserRegAndReset () ;
   CHECK(coretimer_reg.getReadsWrites() = = 4);
   ARRAYS_EQUAL(expected_value, coretimer_reg.getFinalValue(), 4);
}
```

Tests for hardware set-up

The major hardware component involves the need to simultaneously read the asynchronous signals from the two temperature sensors, together with determining the values of two switches used to reset and start the game. There are 4 inputs – programmable flags (PF) – easily accessible on the EZ-kit evaluation board.

The original concept was to use an approach where changes on the PF lines caused interrupts. The switch-driven interrupts caused changes in various flags, allowing the main task to transition between “waiting for the game to start” state, the “game-playing” state and “resetting the game while the game is running or if a winner has been determined” state. The interrupts on the temperature sensor driven lines would be used to access the value of a free-running core timer. This timing information could then be processed to generate the temperature values.

However, using the TDD approach of “easiest way to satisfy the customer requirements”, we decided instead to use a single core

timer driven interrupt rather than 4 PF-driven interrupts. During the timer interrupt routine, we could determine the sense of the input signals and therefore set the necessary flags to transfer between the main task states. We could also simply count the

number of interrupts that occurred between the various high-to-low and low-to-high transitions of the temperature sensors. These counts are sufficient as only a ratio of times, rather than absolute times, as is needed to calculate the temperature. Sufficient precision in the time counts could be guaranteed by having the timer interrupts occur frequently enough.

Generating the timer tests.

Setting up the Blackfin core timer should be a simply enough task, you would think. You need to place the required period (reload value) into the TPERIOD register and the initial count into the TCOUNT register. Next the TSCALE register is set; this determines the relationship between the core timer clock and the 500 MHz system clock. All that remains is to switch the bits in the control register TCNTL so the timer comes out of low-power mode, the automatic reload feature is enabled and the timer is turned on.

All seems pretty straightforward – why bother to write a test? Listing 4 shows my SetupTimer() test and Listing 5, the required Blackfin assembly code needed to satisfy that test. In the test, a WatchDataClass object coretimer_reg is established to watch action occurring amongst the 4 memory mapped core timer registers. The expected registers values after the SetCoreTimer-ASM() function are set up and used for validation.

```
# define SCALE 0
# define PERIOD 0x2000
# define COUNT 0x2000

typedef ulong unsigned long int;

TEST(Test_SetCoreTimer, INSTRUCTOR) {
   // SaveUserRegAndReset () ;
   WatchDataClass<unsigned long> coretimer_reg(4, pTCNTL, pTPERIOD, pTSCALE, pTCOUNT);

   // Setup expected values
   unsigned long expected_value[] = {
      0x0, PERIOD, SCALE, COUNT};

   WATCH_MEMORY_RANGE(coretimer_reg, (SetCoreTimerASM(COUNT, PERIOD, SCALE)),
      READ_CHECK | WRITE_CHECK);
   __SaveUserRegAndReset () ;
   CHECK(coretimer_reg.getReadsWrites() = = 4);
   ARRAYS_EQUAL(expected_value, coretimer_reg.getFinalValue(), 4);
```

Listing 3. The upper code, written the “simplest, obvious way”, does not satisfy all the tests as the cast from floating point occurs at the wrong time and introduces a loss in the required precision. The lower code is a refactored version that provides the required precision and is more appropriate for the integer Blackfin processor.
The WATCH_MEMORY_RANGE macro is called to use hardware breakpoints to record the internal events when activating SetCoreTimerASM(). Finally two checks are made. The first, using the getReadsWrites() method, determines whether the expected number of writes occurs to 4 core-timer registers occurs. The second, using the getFinalValue() method, compares the actual final and expected final values of the core timer registers.

The Blackfin assembly code (Listing 5) is also straightforward to follow, given the “C” like assembly code syntax on the Blackfin processor. First a pointer register is initialized with the address of the required memory mapped core-timer register. Then the function incoming parameters count (in R0), period (in R1) and scale (in R2) are stored in the timer registers.

The old timer control value is stored to be returned when the function exits (in R0) before the timer is enabled in the required mode.

As it turns out, this code sequence, as written, requires the expected four writes to the timer registers, together with an additional read of the timer control register (to capture its initial value) that was not taken into account during the initial test design. Thus the test needs to be rewritten to meet the true SetupTimer() requirements

CHECK(coretimer_reg.getReads Writes() == 5);

as the initial test for only 4 read / writes will fail.

With this change made, the test and code perfectly match, or do they? Listing 6 show the test written by my students as part of an assignment using test driven development to validate their

Test driven development for embedded systems

Copyright M. R. Smith – November 2004
Test driven development for embedded systems

I wrote my assembly code version of SetupCoreTimerASM() in the most obvious way – first parameter into a register, second parameter into a register and so on – a bottom up approach. This meant that the value put into the COUNT register gets over written with PERIOD value. If I had used a top-down approach (third value into a register etc.) or simply had a different order of the parameters in the function prototype, then the COUNT register value would not have been destroyed by the PERIOD value.

This result indicates an unanticipated advantage of the WatchDataClass; the ability to spot undocumented features or changes between different releases of the board or other hardware.

Interrupt and programmable flag input testing.

Given the space limitations, and processor specific characteristics of the code, we will provide neither details for the core timer interrupt service routine to measure the temperature sensors and switch positions nor details of the code to set up and use the digital level programmable flags. However the tests developed for these routines are very revealing on the capabilities, limitations and future directions needed for our current test driven development environment.

Listing 7 demonstrates the test for the basic functionality of the core timer interrupt service routine (ISR). Here a software interrupt is issued using the raise() procedure found within the standard “C” development. A software interrupt occurs each time around a loop. The ISR is supposed to increment a known semaphore number_coretimer_interrupts. The function __SaveUserRegAndReset() provides a call to the VisualDSP C++ runtime environment to place the address of the ISR into the event handler, and unmask the timer interrupt bit in the global IMASK interrupt mask.

It was unclear whether a software interrupt produced by raise() required a corresponding lower() to clear the interrupt and stop re-entrance into the ISR. A CHECK() was therefore added to ensure that the number of times the ISR was entered was equal to the number of calls to raise() -- which should be equal to the number of times around the loop. In hindsight, this is a pointless test. If a call to lower() is in fact required to stop the ISR being re-entered, then this CHECK() statement would never be reached. However the TEST() does demonstrate the functionality of the interrupt service routine.

Developing the test for the functionality of the programmable flag input routines show up the problems that we need to push our current test driven environment to the next level. Now the tests must involve peripherals with externally generated signals, rather than the internal signals provided by the core-timer. One possible solution is to switch the tests away from the real board and place them on the architecturally accurate simulation environment available with the development IDDE. Although the Visual IDDE for the Blackfin ADSP-BF533 supports many peripheral simulations, input on the PF lines is “for a later release”.

A second approach is to “fake our own input signals” by writing known values to the PF input register FIO_FLAG_D and then “pretending” that the signal came from an outside device. However, the FIO_FLAG_D I/O register appears to be like many I/O registers on other processors. If you write a value to a pin that is configured as an input, then the written value is ignored. We still think that we might be able to get around this limitation, but not in the immediate future; meaning before tomorrow (Friday) morning. Thus a different approach to simulating the expected external signals is required.

Listings 8 and 9 show two different approaches we have taken to solve this problem. In Listing 8, the test involves having the developer set up a known input hardware configuration and then test for that input. Straight-forward perhaps, but this approach completely defeats the automated testing procedure that is a major feature of the TDD environment. It is just a too inconvenient approach to be practical in more than a short term.

In Listing 9, an automated test is programmed. Here, the output lines used to control the LED display on the Blackfin evaluation board are looped back to the PF input lines. A background interrupt task involving one of the processor’s timers is used to mimic the TMP03 temperature sensor signal. In this current situation, a self-test was performed using the LED and PF lines on one board. However, for more complex tests, it would be more practical to store the test code in the flash memory of a second Blackfin board. This second board could then be used to generate a variety of video, audio or logic level tests in response to requests from the TDD package on the first board transmitted over the high speed serial port (SPORT), low-speed UART, high speed serial

Copyright M. R. Smith – November 2004
parallel interface (SPI) or through the available multi-processor capability of the JTAG emulation environment.

Having an easy approach to generating these hardware tests is the next step in our research project. We are examining whether the use of the automated software component environment – a typical software engineering design process – can be adapted to generate, or make easily available, tests that can be used across a variety of platforms and design requirements.

In summary

We have described how we have added functionality to a test driven development environment to make it suitable for hardware-software design on small embedded systems. We have demonstrated the capabilities of the system through the design of a “thermal arm wrestling” game suitable for encouraging high school students to enter engineering.

The system provided an automated test environment for embedded systems capable of handling standard code validation, core timer operations including interrupts, and basic I/O. The Analog Devices ADSP-BF533 Blackfin microcontroller had many internal resources to support the hardware side of the TDD. We have found that the current state of this TDD tool has made for easier development and explanation of hardware-based laboratories and projects at the undergraduate and graduate levels on a variety of telecommunications, graphics and video applications. For more information and examples, visit links from the URL http://www.enel.ucalgary.ca/People/Smith/

However, is the approach practical on other embedded systems with less internal resources than the Blackfin? Can we develop a simple automated signal generator than runs on a second embedded system in response to signals from the system under test? Are we getting unnecessarily complicated with the tests, or are the issues discussed simply an indication that very quickly our test driven development environment has moved out of diapers and into the real world?

Copyright M. R. Smith – November 2004
If you would like to see some basic tutorials on test-driven development on embedded systems, or grab our modified source code, please visit the links given on the URL http://www.enel.ucalgary.ca/People/Smith/embeddedTDD/. The authors would be very pleased to hear from you with suggestions of other uses of the current TDD environment. They would be particularly pleased to hear about any solutions to problems that have been raised in this article, or are going to occur in the next stages of the research.

Information about the authors

Mike Smith has been writing computer-related articles since the early ‘70s. He is a professor in Electrical and Computer Engineering at the University of Calgary, Canada. Amongst his interests are the development and testing of algorithms for real-time analysis of stroke images from magnetic resonance imaging. He has received the Analog Devices University Ambassadorship award since 2002.

Andrew Kwan is currently an internship student at the University of Calgary, Canada. He was responsible for most of the development for the WatchDataClass. He will be returning to his 4th year undergraduate studies in Computer Engineering in 2005.

Lily Huang is currently a M. Sc. student at the University of Calgary. She was involved in the initial stages of developing the WatchDataClass. Her research will involve examining theoretical and practical issues with test driven development in an embedded environment.

Moreno Bariffi is an international internship student visiting from the University of Applied Science, Fribourg, Switzerland. He was responsible for integrating the video interface into the project.

Warren Flaman is an electronics technician at the University of Calgary responsible for the implementation details of the thermal arm wrestling external hardware.

Alan Martin, as an internship student, was responsible for making the initial changes to fit the test driven development tool into the confines of an embedded system. He has demonstrated the practicality of using the background telemetry channel as a means of reporting the results of tests without the necessity of stopping the processor. He has returned to his 4th year undergraduate studies in Electrical Engineering at the University of Calgary.

Adam Geras is currently a Ph. D. student at the University of Calgary, Canada. He is combining more than 15 years of industrial experience with an interest in finding the best approaches for adopting and using test driven development schemes in various project and organizational situations.

James Miller is a professor in Electrical and Computer Engineering at the University of Alberta, Canada. In addition to developing small self-contained embedded systems for health monitoring, he has interests in efficient software testing procedures to ensure quality.

Thanks go to Mimi Pichey, Bob Miessenholder and Kamran Sharifi of Analog Devices for hardware, software and other support. Components of this project have been funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada and the University of Calgary. Thanks also go out to the (long-suffering) students in the 3rd year ENCM415 “Assembly Language and Interfacing” course at the University of Calgary who have provided so much useful feedback on the use of TDD in an undergraduate environment.