Unit Testing the MSP430 Within a Desktop Environment

Kris Dickie – B.Tech
Clarius Mobile Health
kris.dickie@clarius.me

I. Introduction

The MSP430™ microcontroller (MCU) is a low-power device designed and produced by Texas Instruments (TI), and comes in a variety of models. The purpose of this technical paper is to explain how to build and run a system for unit testing MSP430 MCU code within a desktop environment. For embedded designs, emulators are often the best choice for testing software, however there may be restrictions such as inability to deploy emulators easily to test systems, cross-platform restrictions, or the complete non-existence of an emulator. This paper describes the setup and gives examples of how to write and execute MSP430 MCU code within a Linux or Windows desktop environment, and has been tested using Google's test suite Google Test (gtest). It's worth mentioning that the examples and framework described make use of some C++11 and standard thread support library built into C++, though most native embedded develop prohibits such use.

II. Testing Architecture

The testing architecture will greatly depend on how your MSP430 MCU code has been structured. For native C applications, a series of function calls may be required to determine an ultimate result, and in a more object-oriented model, a simple constructor initialization and call to one public function may yield the results desired. One of the most important ideas in unit testing is to get as much code coverage as possible, as to run every single line of code at least once, and often multiple times with varying inputs. In reality, this is more difficult to implement, since certain scenarios may be difficult to replicate at a testing level, however by thinking about this up-front in the design of a program, one can help increase code coverage by designing a proper architecture from the beginning.

The architecture described here is a C++ based system that tries to encapsulate hardware modules within instantiated objects. Because of the relatively static nature of embedded programs, all object instances are declared up front and no dynamic memory allocation is used. This is often a good approach to use within environments outside of desktop computing as resources are generally more optimized.

The tests are designed to instantiate objects at any point within the test execution and make appropriate calls to the public member functions of each object.

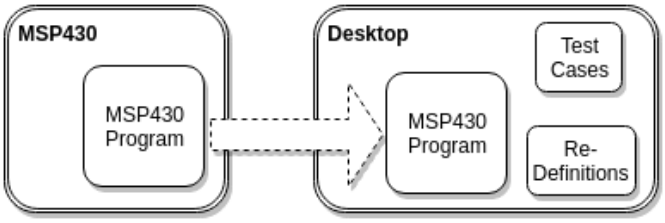

Figure 1. Test Program Migration

III. Working with Registers

As the MSP430 MCU primary I/O functionality is driven through registers, it is important that the testing architecture can properly handle both reads and writes from registers. Since a register is just a memory address, the primary goal of the program will be to write peristent values, readback values, and get notified when a value is changed.

Each register is defined in a specific header related to the MSP430 model (the examples shown are based on the MSP430F5438A MCU), and is either 8, 16, 20, or 32 bits in size. A typical register

definition would look as follows:

```
SFR_8BIT(ADC12CTL0_L);
```

The key for register definition in the testing program is to redefine the macros that initialize each register; this can be achieved by referencing an externally defined type that encapsulates the idea of a register. In this case, a template works nicely to capture the different register sizes.

```
#define SFR_8BIT(address) /
extern testReg<unsigned char> address

#define SFR_16BIT(address) /
extern testReg<unsigned int> address

typedef void (* __SFR_FARPTR)();
#define SFR_20BIT(address) /
extern __SFR_FARPTR address

#define SFR_32BIT(address) /
extern testReg<unsigned long> address
```

Now when including the msp430<model>.h header file for register and bitmask definitions, the macros will have a proper definintion to reference.

The `testReg` reference is a templated class which allows the program to define registers of different sizes (char, int, and long for example), and is derived from the C++ class `std::condition_variable` which allows each register to individually notify any thread that is waiting for it's value to change.

The definition of all template classes need to be in a header file in order for them to compile.

```
template <typename T>
class testReg : std::condition_variable
{
public:
    testReg() : val_(0) { }
    testReg(T val) : val_(val) { }

T wait()
{
std::unique_lock<std::mutex> lk(lock_);

std::condition_variable::wait(lk);
return val_;
}
testReg& operator=(T val)
{
    val_ = val;
    notify_one();
    return *this;
}
testReg& operator|=(T val)
{
    val_ |= val;
    notify_one();
    return *this;
}
testReg& operator&=(T val)
{
    val_ &= val;
    notify_one();
    return *this;
}
T* operator&()
{
    return &val_;
}
operator T() { return val_; }

private:
    T val_;
    std::mutex lock_;
};
```

The `testReg` class creates two member variables, `val_` which holds the current register value (initialized to 0), and `lock_` which is used for thread synchronization when an outside value is waiting for a register value to change.

The class also makes use of operator overloading to handle direct and bitwise assignments, when these are used from within the MSP430 MCU program, a call to `notify_one()` will notify any thread waiting on that register value to change, so that it may read back the value or continue it's execution.

Example:

```
void thread1()
{
    P1OUT = 0;
    // start thread 2
```

```
    P1OUT.wait();
    int val = P1OUT;
    // value should be 1
    printf("P1: %d", val);
}

void thread2()
{
    P1OUT |= 0x01;
}
```

There is one final step needed for a program to compile. Since we've defined each MSP430 register as an extern testReg, the actual definition must be created, i.e. the header file is not good enough, and your program will yield undefined reference errors. This can be accomplished by writing a simple python script to parse the MSP430 header file that includes the registers definitions and create an implementation file (i.e. .cpp file) that define each register. The script needs to parse out all SFR_XBIT definitions in the MSP430 header file and create a file that will look similar to the following:

```
#include <msp430.h>

testReg<unsigned char> ADC12CTL0_lL (0);
...
testReg<unsigned int> WDTCTL (0);
```

Where each platform will have slightly different register configurations. With this file generated, and included in the testing project, the registers now have proper definitions.

IV. Overriding Intrinsics

It may be important for the testing program to override specific MSP430 MCU intrinsic calls and is much simpler than trying to get native assembly to compile and run within a desktop environment.

The simplest approach is to simply create an implementation file (i.e. .cpp file) that defines a specified operation for the instrinsics that are used within the actual MSP430 MCU program that is being tested. For example:

```
#include <msp430.h>

void __data20_write_char(unsigned long
addr, unsigned char v)
{
    *((unsigned long*)addr) = v;
}

void __delay_cycles(unsigned long cycles)
{
    usleep(cycles);
}
```

In the above functions, we simply implement a write to an address for __data20_write_char(), and implement an actual wait time for __delay_cycles().

One caveat that potentially needs to be addressed is the use of defined types and address spaces. In the example __data20_write_char(), unsigned long will be defined as a 32-bit number on the MSP430 MCU, and will likely be on a desktop environment as well, as unsigned long long may represent a 64-bit address. Now since __data20_write_char() is converting the input into an actual pointer, dereferencing it, and stuffing a value, this could potentially lead to issues and crashes during testing on a 64-bit operating system, where the memory space is beyond the range of unsigned long. One must remember that a call to __data20_write_char may yield an input address anywhere in memory space on the desktop.

A solution to this problem is to define the __data20_write_char() function with a uint64_t address type; this is easily accomplished for the implementation, but the default definition in the intrinsics.h would also have to change in order for compilation to succeed. Again, a python script can be used to update this definition, or creating a new intrinsics.h file for testing can be performed as well, since it's contents is relatively small.

V. Flash Memory Testing

Testing flash memory reads and writes may be useful for an application, especially if critical data

is stored within information banks. Depending on the platform, the number of information banks will vary, however it will be important to define new addresses as opposed to using BANKA, BANKB, etc. which are MSP430 MCU specific. The MSP430 program should invoke a method to set variable addresses at runtime, which are then referenced for all reads and writes. The default implementation would set addresses to the BANK definitions, and the test program would then be able to make a subsequent call to set addresses from a defined memory space that was setup on the desktop environment.

Since a flash memory write typically requires waiting for register bits (i.e. WAIT or BUSY) to be set from the device, the test program must ensure that these bits can be set when required, otherwise the testing will hang. For most cases, writing a thread which asserts the WAIT bit, and de-asserts the BUSY of the register FCTL3 should do the trick, once the LOCK bit is asserted on the same register, the thread can exit.

As an example:

```
void setFlash()
{
  while (!(FCTL3 & LOCK))
  {
    FCTL3 &= BUSY;
    FCTL3 |= WAIT;
  }
}

void testFlash()
{
  FCTL3 = 0;
  std::thread t([&]{ setFlash(); });

  mspfunc::writeFlash(address, data);
  rd = mspfunc::readFlash(address);
  ASSERT(rd == data);

  t.join();
}
```

The exact implemenation may differ somewhat, but the example is meant to show how a thread can ensure the proper bits are set during an operation that would take place during standard MSP430 MCU operation through the calls to the namespace mspfunc.

VI. Communication Buses

Communication buses can sometimes be more challenging to test because of their reliance of an outside source generating an interrupt. One design approach that can be used is to write classes with a public method to call the interrupt routine, or in the case of a C program, a global function that can call the interrupt directly. We can look at the SPI bus as a starting point.

```
class spibus
{
  public:
    void testInterrupt() { onRxData(); }

  private:
    static __interrupt void onRxData();
    char buffer[32];
};

#pragma vector=USCI_A2_VECTOR
__interrupt void spibus::onRxData()
{
  switch (__even_in_range(UCA2IV, 4)
  {
    case 2:
      // fill buffer
      *buffer++ = UCA2RXBUF;
      // exit low power mode to allow
      // buffer to be processed
      __bic_SR_register_on_exit(...);
      break;
  }
}
```

This now allows the interrupt to be called by an outside testing thread.

To take the example further, a thread can now be used to virtually send data through the SPI bus, while the test program waits for a buffer to be filled.

```
spibuf spi;
```

```
void sendData(std::vector<char>& buf)
{
  UCA2IV = 2;
  for (auto i = 0u; < buf.size(); i++)
  {
    UCA2RXBUF = buf[i];
    spi.testInterrupt();
  }
}

std::vector<char> buf = “test”;
std::thread t([&]{ sendData(buf); });

while (!spi.finishedParsingBuffer());

t.join();

ASSERT(spi.buffer == buf);
```

This is just one potential method to try and synchronize the SPI read, another could be to actually wait for the receive buffer to receive a value, and call a separate parsing function similar to the following:

```
int count = 0;
while (count++ < buf.size());
{
  UCA2RXBUF.wait();
  spi.parseNewData();
}
```

To test the full response chain on a SPI bus, with an object waiting for messages, parsing them, and then transmitting them back. For instance if there was a query for the battery level of a device, we can use the above examples to implement the receive side to respond to the request, and then monitor the SPI transmissions in a separate thread.

```
void monitor(int sz, std::vector<char>&
buf)
{
  while (buf.size() < sz)
  {
    UCA2TXBUF.wait();
    buf.push_back(UCA2TXBUF);
  }
}

std::vector<char> tx;
std::vector<char> rx = “battery”;
spi.setBatteryLevel(100);
std::thread t1([&]{ monitor(1, tx); });
std::thread t2([&]{ sendData(rx); });

// assume the parsing function will
// respond internally by sending the
// battery level as 1 byte over SPI
while (spi.parsing());

t1.join();
t2.join();

ASSERT(tx == 100);
```

Using similar methods can be used to test the I2C or UART bus, as well as ADCs. Each will have their own nuances that need to be addressed, for instance, I2C tests will likely have to deal with setting of start and stop flags, and possibly handle NACKs sent as an interrupt.

VII. Simulating Timers

As with communication buses, timers can also be somewhat difficult to test because of their reliance on a temporal interrupt.

There are many C++ constructs for timers, but one of the best for high-resolution timing lies within the boost::asio libraries, specifically the deadline_timer.

To setup a system that will generate timer interrupts, a deadline_timer can be instantiated with a specific interval that is based on a capture/compare register. An example for Timer1_A3 running, capture/control register 0.

```
class msptimer
{
public:
  void testInterrupt() { onTick(); }

  void init(int frequency)
  {
    // perform setup
    TA1CCR0 = (ACLKFREQ / frequency) + 1;
  }

  void start()
  {
    TA1CTL |= MC__UP;
```

```
  }
  void stop()
  {
    TA1CTL &= ~MC__UPDOWN;
    TA1R = 0;
  }

private:
  static __interrupt void onTick();
  int ticks;
};

#pragma vector=USCI_A2_VECTOR
__interrupt void msptimer::onTick()
{
  tick++;
}
```

There is now a class that can run a timer with the implementation details left to be determined, except that the interrupt will increment a counter. To setup the test system that will actually call the interrupt, the following deadline_timer setup can be used.

```
void setup(const msptimer& t)
{
  // setup the timer based on the
  // initialization of the
  // control/capture register
  int val =
   (TA1CCR0 / ACLKFREQ) * 1000000;
  boost::asio::io_service ios;
  boost:posix_time::microseconds
   ival(val);
  boost::asio::deadline_timer
   tmr(ios, iva);

  // wait for the msp timer to start
  TA1CTL.wait();
  TA1CTL |= MC_3;

  // start the deadline_timer, call
  // btick  on an expiration
  tmr.async_wait(boost::bind(btick, ...,
   t, &tmr, ival);
  ios.run();
}

void btick(const msptimer& t, tmr, ival)
{
  t.testInterrupt();
  // if timer still running, then
  // reschedule the deadline_timer
  if (TA1CTL & MC_3)
  {
    tmr->expires_at(tmr->expires_at() +
     ival);
    tmr->async_wait(...);
  }
}

// create timer running at 1kHz
msptimer t;
t.init(1000);

// setup the deadline_timer
std::thread t([&]{ setup(t); });

// run timer, and wait for 1 second
t.start();
sleep(1);
t.stop();

ASSERT(t.ticks ≈ 1000);
```

The assertion of the number of ticks elapsed will be approximate, as the sleep() function and setup times are not necessarily exact, but the high resolution of a boost deadline_timer should make for a relatively precise way of measuring time if the capture/control registers are used for setting up the timer interval.

VIII. Thread Synchronizatoin

One important concept that has been left out of the previous examples, mostly for brevity, is the use of thread synchronization to ensure that the test program does not hang and that register values get set at the appropriate times.

In the case of the previous example of setting up a SPI communications bus for testing, there is a worker thread will generate interrupts for the SPI receive. To ensure that the worker thread does not overtake the main thread, without the proper setup being put in place, it is required that proper wait conditions are used. The example can be rebuilt with notations on how synchronization is performed. Specifically, a condition variable, a mutex, and a unique lock are typically required in order to perform proper synchronization.

```
// define a short-hand wait macro
#define wait_usec(s) /
  std::this_thread::sleep_for /
  (std::chrono::microseconds(s));

// function to send SPI data by loading
// the RX buffer and generating
// test interrupts
void sendData(std::vector<char>& buf,
  std::condition_variable& cv)
{
  // ensure the interrupt vector is set
  // for receive
  UCA2IV = 2;

  // short delay to allow main thread
  // to call it's wait function
  wait_usec(1);
  // notify main thread that we're ready
  cv.notify_one();
  // send out the data
  for (auto i = 0u; < buf.size(); i++)
  {
    UCA2RXBUF = buf[i];
    spi.testInterrupt();
    // allow time for the interrupt to
    // be processed, typically a monitor
    // of a buffer from the main thread
    wait_usec(10);
  }
}

// main thread and main entry point
void mainThread()
{
  // setup the synchronization objects
  std::condition_variable cv;
  std::mutex mtx;
  std::unique_lock<std::mutex> lk(mtx);

  // initialize other items required
  // for the comm bus

  // start the worker thread to send data
  std::vector<char> buf = "test";
  std::thread t([&]{ sendData(buf,
   cv); });
  // ensure the thread is ready
  cv.wait(lk);

  // parse the buffer as it fills up
  // with data
  while (!spi.finishedParsingBuffer());

  // ensure the worker thread is finished
  t.join();

  // check result
  ASSERT(spi.buffer == buf);
}
```

Condition variables can be used in a worker thread as well when both threads need to setup a pre-determined state before the full testing loop can begin; this can be done simply by passing in additional information such as the mutex and unique lock object into the worker thread as well.

IX. Conclusion

MSP430 MCU programs can be written for a multitude of applications, and may use simple or complex algorithms and logic, but as for all software development, having a testable application can help ensure reliability and performance no matter what the application. By making use of some basic C++ constructs, migrating an application to be tested on the desktop can be relatively painless and provide a flexibile environment to help implement tests that provide more code coverage and execute functions that would traditionally only be possile on the embedded device itself.

About the Author

Kris studied Computer Systems at the British Columbia Institute of Technology, and has since been working in the medical device industry for over 16 years, helping to design and implement real-time imaging devices using a variety of platforms and technologies, which include: Microsoft Windows, Embedded Linux (ARM based), TI C6X DSPs, TI MSP430, ADI Blackfin SoC, and Xilinx FGPAs.