CSC 460 Project 2

Building A Real Time Operating System

Geoffrey Gollmer - V00730142
Adam Anderson - V00724357

1.0 Introduction

In project 2 we were tasked with taking an existing RTOS and updating it to run on the ATmega2560 board, including context switching and error display. After the operating system was ported over, we were tasked with improving its task scheduling system. The existing system contained a set of round robin tasks, each with a noted worst case execution time (WCET). The new implementation includes 3 types of tasks. In order of priority: System tasks, executed in order of creation, Periodic tasks, each with a recorded worst case execution time, and finally round robin tasks, which run for 1 tick (5ms) before swapping to the next task. Task preemption based on priority and WCET extensions due to preemption were also required.

Additionally, we were to add a publish/subscribe service to the system. Subscribing tasks are paused until another task publishes to the subscribed service. Published values are a simple 16 bit int designed to represent messages between tasks.

Finally, a Now() function was to be added to the operating system to provide accurate millisecond timing for tasks.

2.0 Hardware

The real time operating system runs on an arduino board using an ATmega2560 processor. The board has 6 timers (2 8-bit, 4 16-bit), one of which is reserved by the RTOS (timer1 16-bit), 52 digital I/O pins, and 16 analog I/O pins. The ATmega2560 processor is a RISK based processor that runs at 16Mhz, with 8KB of ram and 64KB of flash memory, running on 2.7V - 5.5V of electricity.

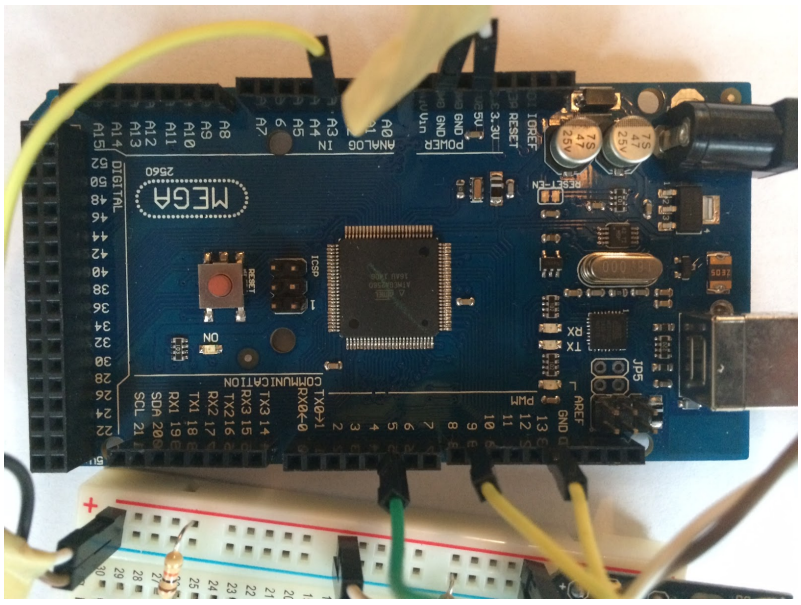

Figure 1: ATmega2560 board

The operating system is tested using the board's built in LED (controlled by port 13), and an 8 bit signal analyzer.

Figure 2: Saleae Digital Signal Analyzer

3.0 OS Setup

The provided OS was not capable of compiling or running on the ATmega2560 board. In order to port the OS to the new board, a few changes had to be made. The first was fixing actual compilation errors in the source code itself. These errors consisted mostly of typos or changed constant names whose references hadn't been updated. Once the OS was capable of compiling, it had to be pushed onto the board, and finally the kernel level context switching code had to be changed to suit the memory reference size of the ATmega2560.

3.1 Building for the ATmega2560

In order to build and compile for the ATmega2560 we used a product called CrossPack for AVR[1] which contains AVRDUDE for uploading and a few useful tools for compiling code. With CrossPack installed, we wrote a shell script to automate the process of compiling, linking, and uploading our operating system. The build script accommodates for the processing speed of the board, generates a hex file to run on the board from the compiled elf file, and finally uses avrdude to upload the code to the board itself.

```
echo "Clean..."
rm -f *.o
rm -f *.elf
rm -f *.hex

echo "Compile..."
avr-gcc -Wall -O2 -DF_CPU=16000000UL -mmcu=atmega2560 -c main.c -o main.o
avr-gcc -Wall -O2 -DF_CPU=16000000UL -mmcu=atmega2560 -c os.c -o os.o

echo "Link..."
```

```
avr-gcc -Wall -O2 -DF_CPU=16000000UL -mmcu=atmega2560 -o main.elf os.o main.o

echo "Make HEX..."
avr-objcopy -j .text -j .data -O ihex main.elf main.hex

echo "Uploading..."
sudo avrdude -p m2560 -c wiring -P /dev/tty.usbmodem1411 -U flash:w:main.hex:i
```

Snippet 1: build.sh

3.2 Porting Existing RTOS to ATmega2560

The provided RTOS for project 2 was designed to run on the ATMega1280 which uses 128K of program memory. Since we are running on the ATMega2560 which uses 256K of program memory, some modifications had to be made to the system in order to support the new hardware.

Aside from a few compilation and syntax errors in the existing code, the primary problem in porting the code to the ATMega2560 was dealing with different memory address lengths. The ATMega1280 uses 16 bits to address program memory, which is the exact size of a register within the board. The ATMega2560 however has to use 17 bits for memory references. The 17th bit comes from a third register known as EIND, and determines which of two sections of 128K of memory is used. In order to support proper context switching, this 17th bit had to be stored in memory and restored correctly during a context switch. Additionally, the size of function pointers increases from 2 bytes to 3 bytes, changing the expected stack size for tasks within the RTOS.

Figure 3: ATmega2560 address registers

In order to fix this issue, existing arm code had to be customized to support the 17th bit, as well as the stack creation code for new tasks, in order to ensure memory is available for the extra bit used in function pointers. The implementation we used was based on work done in the summer 2013 CSC 460 class by Mike Krazanowski [2].

4.0 Periodic Scheduler

The primary addition to the existing RTOS consists of a new periodic task scheduling system. The tasks are created by the programmer with a given period, worst case execution time, and start delay, all in ticks. The periodic task must complete its execution within the provided

WCET. No two periodic tasks can run at the same time, so if one task becomes available while another is running, it is considered a runtime error. Periodic tasks are higher priority than the existing round robin tasks, and lower than system tasks. If a system task preempts a running periodic task, the periodic task must hand over control, and it's WCET is extended by the approximate runtime of the system task.

In order to implement this system, a number of parameters were added to the RTOS's task description.

```
/**
 * @brief All the data needed to describe the task, including its context.
 */
struct td_struct
{
    /** The stack used by the task. SP points in here when task is RUNNING. */
    uint8_t                         stack[MAXSTACK];
    /** A variable to save the hardware SP into when the task is suspended. */
    uint8_t*               volatile sp;   /* stack pointer into the "workSpace" */
    /** PERIODIC tasks need a known period. */
    uint16_t                        period;
    /** PERIODIC tasks need a known worst case execution time. */
    uint16_t                        wcet;
    /** PERIODIC tasks have a countdown until the next time they run. */
    int16_t                        countdown;
    /** The state of the task in this descriptor. */
    task_state_t                    state;
    /** The argument passed to Task_Create for this task. */
    int                             arg;
    /** The priority (type) of this task. */
    uint8_t                         level;
    /** A link to the previous task descriptor in the queue holding this task. */
    task_descriptor_t*              prev;
    /** A link to the next task descriptor in the queue holding this task. */
    task_descriptor_t*              next;
    /** A link to the value to store service posts in. */
    int16_t*                        value;
};
```

Snippet 2: RTOS Task Description

Values describing the tasks period and WCET were added first. A countdown value was also added. The countdown contains the number of ticks before the periodic task must be executed again. Each tick all periodic tasks' countdown values are reduced by 1. When a task begins running, it's countdown is reset to match it's period. On initialization, countdown contains the programmer's start delay value. Additionally, a prev value was added in order to allow for the creation of linked lists for periodic tasks. The linked list structure is used as a simple list for iterating over periodic tasks when updating countdowns.

4.1 Error Handling

There are a number of possible errors that can occur with periodic tasks that should cause the RTOS to abort and display a corresponding error message. Error messages are encoded using a simple number system. The board's built in LED flashes one long pulse followed by n

short pulses for a setup error, and two shorter pulses, followed by a low value before n more short pulses for a runtime error. Three main errors exist for periodic tasks: impossible periods, running past their WCET, and collisions.

Impossible periods are simple to handle, if a task is created with a WCET longer than its period, the task is impossible to run, and the OS will immediately abort.

Tasks running longer than their WCET are a little more complicated. When a new periodic task begins, a value is set to the task's WCET and reduced each time a tick completes. If the value reaches 0 while the task is still running, the operating system is aborted and the correct error code is displayed.

```
/**
 * @fn kernel_find_periodic
 *
 *@brief Searches running periodic processes for a processes ready to start.
 *
 * Searches the linked list of periodic processes, returns null if none are ready to run.
 */
static task_descriptor_t* kernel_find_periodic(void)
{
    task_descriptor_t* ret_val = NULL;
    task_descriptor_t* periodic_task = periodic_list.head;
    while(periodic_task != NULL)
    {
        if(periodic_task->countdown <= 0) {
            if(ret_val != NULL) {
                error_msg = ERR_RUN_6_PERIODIC_TASK_COLLISION;
                OS_Abort();
                return NULL;
            }
            ret_val = periodic_task;
        }
        periodic_task = periodic_task->next;
    }
    return ret_val;
}
```

Snippet 3: Periodic task scheduling and collision

Finally, collisions are detected when the countdown value for each task is reduced. If a task reaches countdown 0 and another periodic task is currently running, the OS simply aborts and displays the corresponding error message.

These three cases cover the possible periodic errors, and were simple to implement without the concept of preemption.

4.2 Preemption

Preemption occurs when a higher priority task is created or resumed in a queue and needs to steal the current tick from the current task. The primary case for preemption is system tasks

starting or resuming during a periodic task. When this happens, control is immediately taken away from the periodic task and given to the system task. In this case, periodic tasks still need to stay on schedule, but the preempted task should be given an extension on its WCET. Ideally the WCET would be increased by the length of the preempting system task, however that requires a very high granularity in timing. Instead as a simple solution, 1 tick is added to the WCET of the preempted task when the preemption occurs, and the number of ticks remaining is not reduced until the system task completes. This solution covers all scenarios were system tasks run 1 tick or longer, but fails to cover the potential of multiple preemptions occurring inside a single tick.

In order to ensure the preempted task is resumed once the system task completes, its countdown is reduced by its period, becoming a negative number. This makes the periodic task the highest priority task when the system task completes, and resumes it from its previous state.

5.0 Services

Services provide a form of communication between tasks. A single service only contains a list of subscribed tasks, however when a task subscribes to a service, it is paused until a value is published. Subscribed tasks are removed from their respective queues and placed in a waiting state. Since periodic tasks exist on a very specific schedule, they are not allowed to subscribe to a service, but they may publish to one. Additionally, interrupts may publish to a service. The RTOS supports a maximum of 8 services, which are stored as a global array in memory.

5.1 Subscribing

Subscribing is a simple operation. When a task subscribes to a service, its state is set to waiting, the task is added to the service's subscriber list, and the memory to place the published value in is stored in the task's own descriptor. Little to no additional memory is required to create a service this way.

```
void Service_Subscribe( service_t *s, int16_t *v )
{
    if(cur_task->level == PERIODIC) {
        error_msg = ERR_RUN_7_PERIODIC_TASK_SUBSCRIBED;
        OS_Abort();
    }

    enqueue(&(s->subscribers), cur_task);
    cur_task->state = WAITING;
    cur_task->value = v;

    Task_Next();
}
```

Snippet 4: Task Subscription

5.2 Publishing

When a task publishes to a service a single value is sent to each task, and placed in a memory location specified when the tasks subscribe. The tasks are placed into a ready state, and pushed onto the front of their respective queues. The publishing task may choose to release control to the resumed tasks, unless they preempt the publishing task. In this case, the same preemption logic from periodic tasks is used. In order to implement this logic, a new kernel command was created to interrupt the currently running task.

```
case TASK_INTERRUPT:
        if(cur_task->state == RUNNING)
        {
            if(cur_task->level != SYSTEM)
            {
                cur_task->state = READY;
                if(cur_task->level == PERIODIC) {
                    cur_task->countdown -= cur_task->period;
                    ticks_remaining++;                }
                else {
                    push_queue(&rr_queue, cur_task);
                }
            }
        }
        break;
```

Snippet 5: Kernel task interrupt command

5.3 Error Handling

Three major errors can occur when working with services. The first is that the system simply runs out of services to allocate. The other two revolve around periodic tasks attempting to subscribe to a service. The first occurs when a periodic task actually attempts to subscribe to a service. The second is if a periodic task is found subscribed to a service. The second of which should never occur, however it is included in our RTOS for sanity's sake. Handling of each error consists primarily of a trivial if statement.

6.0 Now

Implementing the now function was a simple matter of checking the timer used to iterate over ticks. The timer is cumulative, and only resets when it overflows, however by simply updating the previous tick time each time a new tick starts, some simple math can be used (knowing the timer prescaler and and CPU speed to figure out how many milliseconds have passed since the last tick). By keeping a tick count, the current time can be calculated on a per millisecond basis, and potentially to even more accuracy. In order to keep Now() as fast as possible, the division operator was avoided, instead using simple integer comparisons.

```
/**
```

```
 *  @Brief return time since operation began in millis.
 */
uint16_t Now()
{
    uint16_t retval = current_tick_multiplied-5;
    uint16_t cur_time = TCNT1 - previous_tick_time;
    if(cur_time < MS_CYCLES)
        return retval;
    if(cur_time < MS_CYCLES2)
        return retval+1;
    if(cur_time < MS_CYCLES3)
        return retval+2;
    if(cur_time < MS_CYCLES4)
        return retval+3;
    return retval+4;
}
```

Snippet 6: Now function

7.0 Testing

In order to prove that the implemented RTOS was operating as expected a series of unit tests were created. These unit tests covered all implemented features within the RTOS and all possible errors that could produced. In order to correctly test the RTOS a digital signal analyzer was used to track the status of 8 different digital signals mapped on the board. Tests use a special header that makes toggling these signals trivial. By looking at the output of these 8 pins, the behaviour of the OS can be observed without breaking any of its timing with additional complex commands, like the built-in Serial interface.

7.1 Digital Signal Analyzer

The digital signal analyzer is a Saleae compliant device that simply samples the 8 pins, at an extremely high rate, and transfers the results back to a program running on a computer for analysis[3]. The device can been seen in Figure 2. In order to easily use this system a bridge of 8 pins was created between the board and the device, and a specific header file was designed to abstract the pin mapping used for debug.

```
// Use primarily for debugging. Maps ports to a digital analyzer
// Port0 maps to Pin 30 on the ATMega2560
// Port7 maps to Pin37 on the ATMega2560

#define USE_MAP 1

#define PORT_DATADIRECTIONREGISTER  DDRC
#define PORT_OUT                                PORTC
#define PORT_PIN0                   PC0  // 37
#define PORT_PIN1                   PC1  // 36
#define PORT_PIN2                   PC2  // 35
#define PORT_PIN3                   PC3  // 34
#define PORT_PIN4                   PC4  // 33
#define PORT_PIN5                   PC5  // 32
#define PORT_PIN6                   PC6  // 31
#define PORT_PIN7                   PC7  // 30
```

```
#define DefaultPorts() { PORT_DATADIRECTIONREGISTER |= 0xff; }

#if USE_MAP

#define EnablePort(a) { PORT_OUT |= (_BV(a)); }

#define EnablePort0() EnablePort(PORT_PIN0);
#define EnablePort1() EnablePort(PORT_PIN1);
#define EnablePort2() EnablePort(PORT_PIN2);
#define EnablePort3() EnablePort(PORT_PIN3);
#define EnablePort4() EnablePort(PORT_PIN4);
#define EnablePort5() EnablePort(PORT_PIN5);
#define EnablePort6() EnablePort(PORT_PIN6);
#define EnablePort7() EnablePort(PORT_PIN7);

#define DisablePort(a) { PORT_OUT &= ~(_BV(a)); }

#define DisablePort0() DisablePort(PORT_PIN0);
#define DisablePort1() DisablePort(PORT_PIN1);
#define DisablePort2() DisablePort(PORT_PIN2);
#define DisablePort3() DisablePort(PORT_PIN3);
#define DisablePort4() DisablePort(PORT_PIN4);
#define DisablePort5() DisablePort(PORT_PIN5);
#define DisablePort6() DisablePort(PORT_PIN6);
#define DisablePort7() DisablePort(PORT_PIN7);

#define PortBlip(a) {              \
    EnablePort(a);                 \
        volatile count = 0;            \
        for( int i = 0; i < 2; i++ )   \
        {                              \
                count++;                   \
        }                              \
    DisablePort(a);                \
}

#define PortBlip1() PortBlip(PORT_PIN0)
#define PortBlip2() PortBlip(PORT_PIN1)
#define PortBlip3() PortBlip(PORT_PIN2)
#define PortBlip4() PortBlip(PORT_PIN3)
#define PortBlip5() PortBlip(PORT_PIN4)
#define PortBlip6() PortBlip(PORT_PIN5)
#define PortBlip7() PortBlip(PORT_PIN6)
#define PortBlip8() PortBlip(PORT_PIN7)

#endif
```

Snippet 7: Debug port mapping

7.2 Logic Test Cases

A set of unit tests were designed to make sure the features of the RTOS work as expected. These tests should never result in an error, and should produce their exact expected output.

7.2.1 Sanity Test

The first test simply creates a round robin task that toggles port 0 every 4 milliseconds. Since the task is resumed immediately whenever a new tick begins, it should appear to be running without interruption.

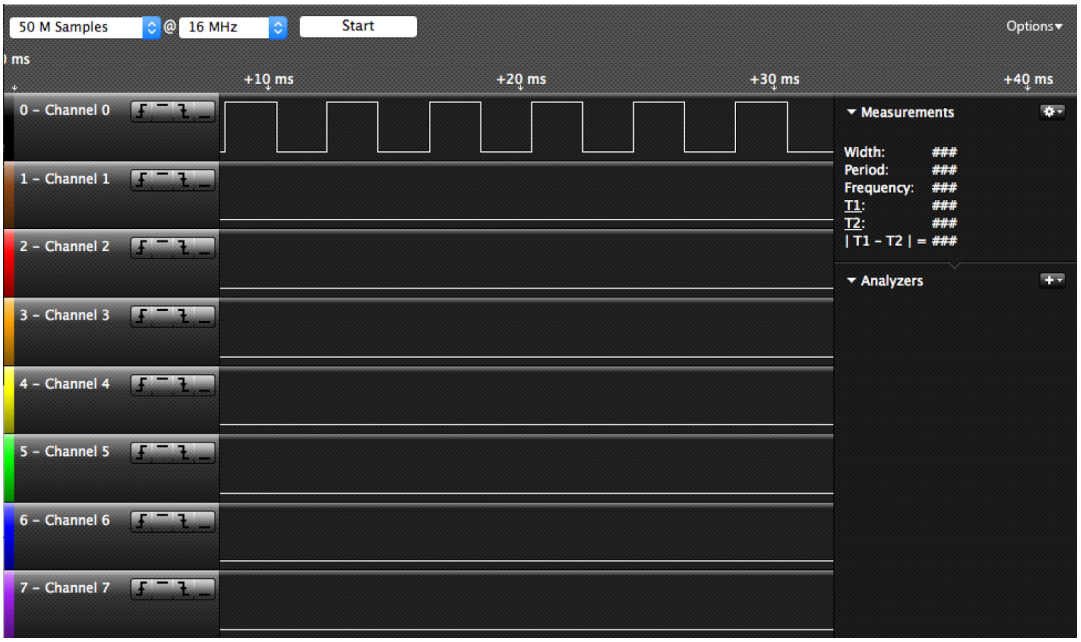

Figure 4: Sanity Test Results

7.2.2 Round Robin Test

The round robin test consists of 3 round robin tasks. Each is in charge of a different channel. When a round robin task is running, it sets its own channel to on and all others to off. Each round robin should run for 5ms before moving on to the next one. Round robin tasks should run in a round robin pattern.

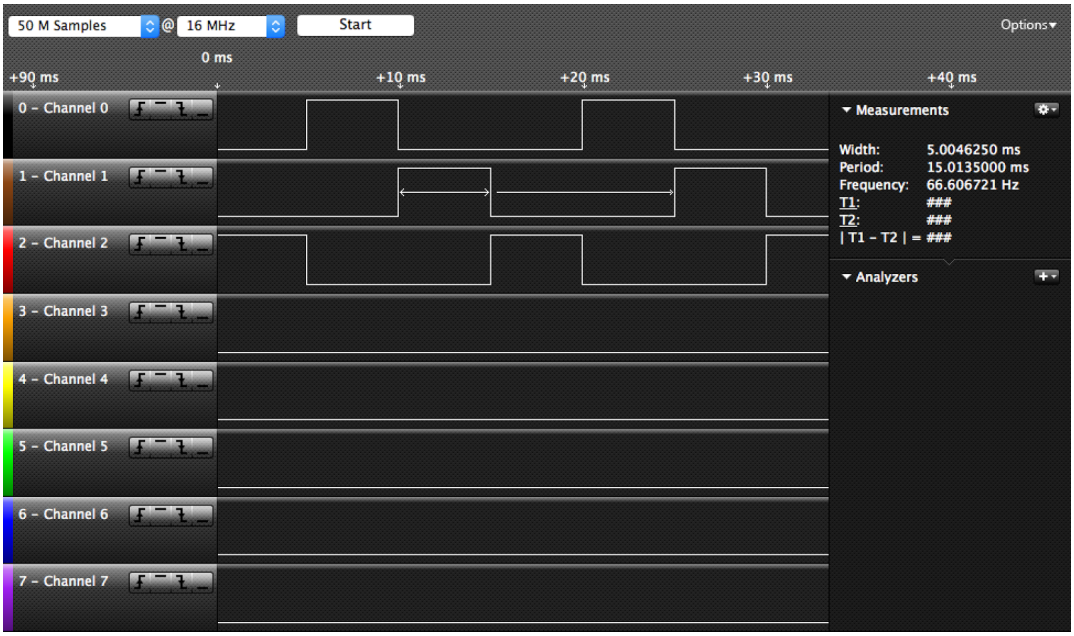

Figure 5: Round Robin Test Results

7.2.3 Periodic Test

The periodic test creates a round robin task that spikes channel 0 every 1ms while running. In addition it creates two periodic tasks, one in charge of channel 1 and one in charge of channel 2. The first periodic task has a period of 5 ticks and runs for 1 tick. The second is the same with a start delay of 3 ticks.

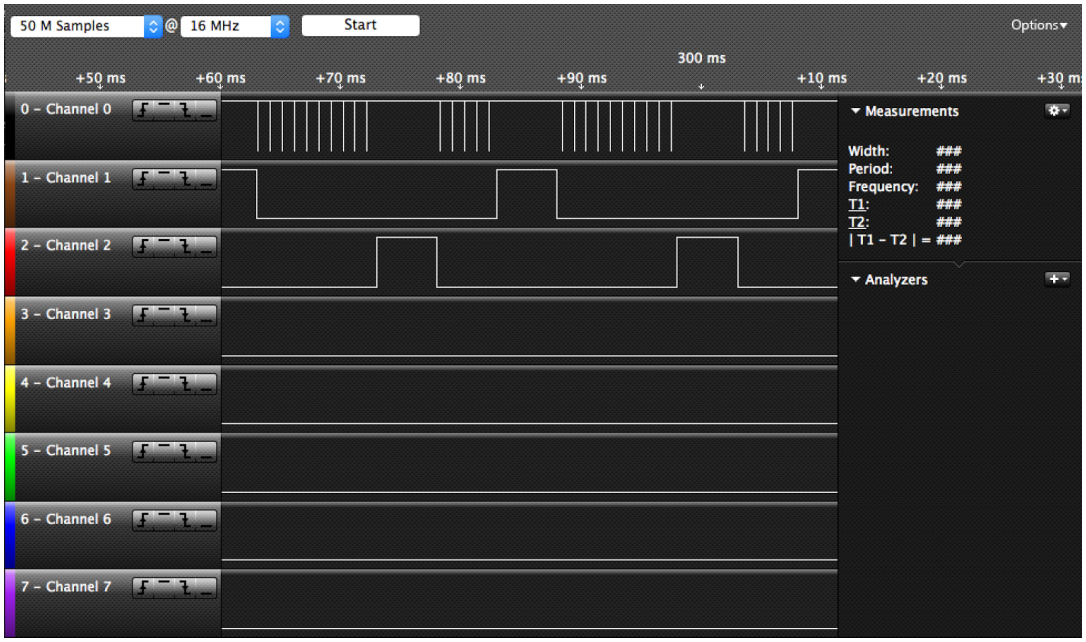

Figure 6: Periodic Test Results

7.2.4 System Task Test

The system task test creates a round robin that spikes channel 0 every 1 ms. Then a periodic task that runs every 6 ticks and has an execution time of 2 ticks. 1 tick into the periodic task, a system task is spawned that runs for one tick before returning to the periodic task. Since this system task preempts the periodic task, the periodic task is paused during execution, causing the periodic task to take 3 ticks to run fully.

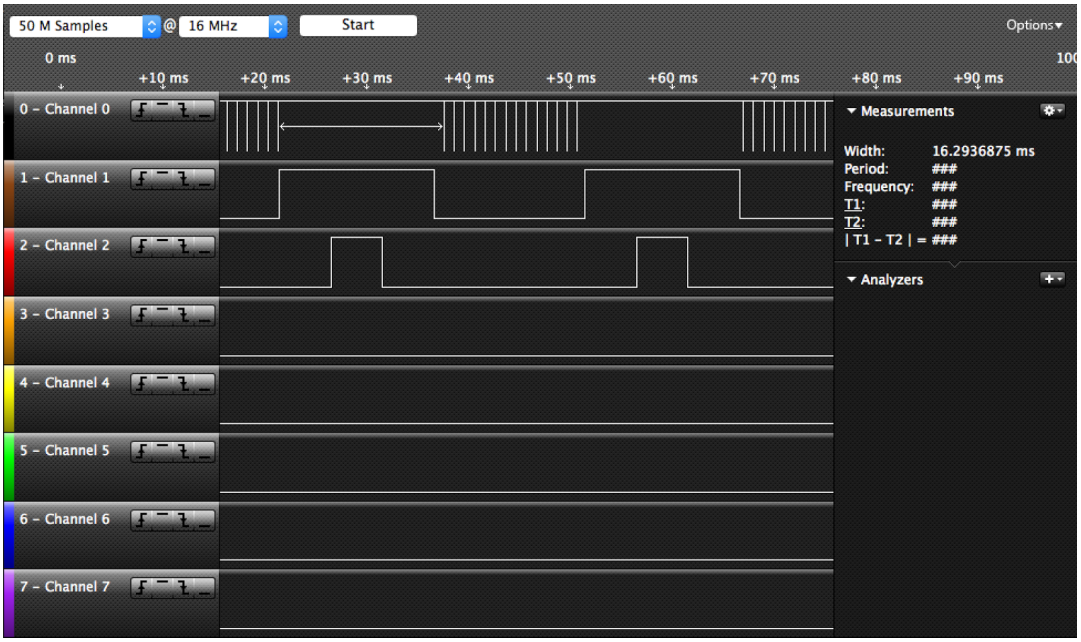

Figure 7: System Task Test Results

7.2.5 Now Test

The now test simply turns channel `current_time%5` on and all others off. This way it can easily be seen the Now() returns the correct time.

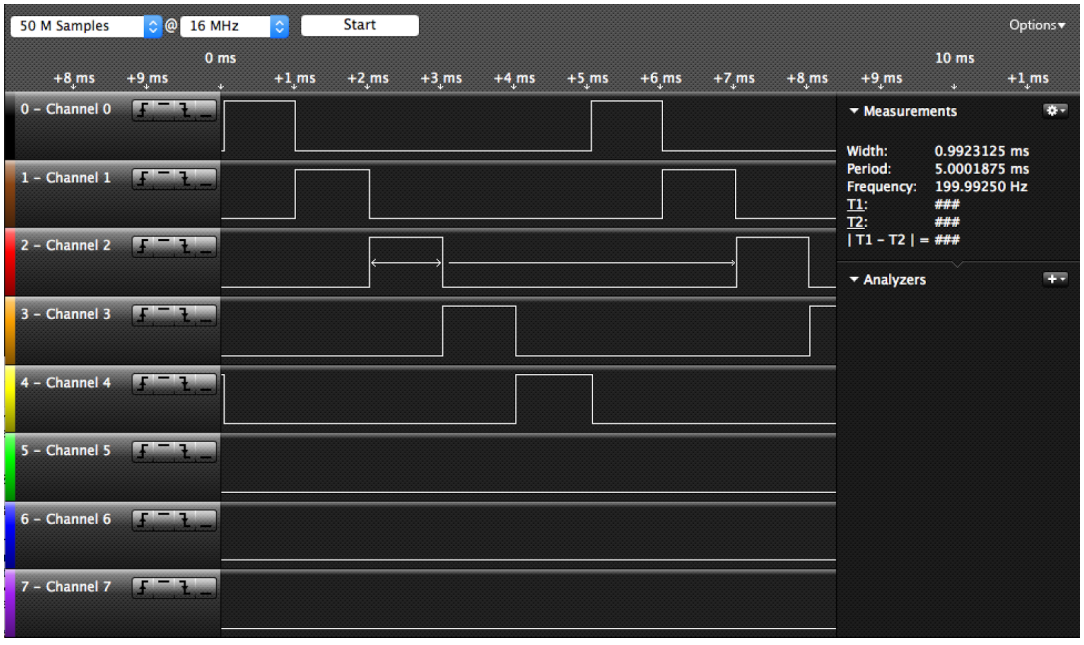

Figure 8: Now Test Results

7.2.6 Task Arguments Test

The task arguments test is complicated. It is designed to test all 3 task types to ensure that arguments are received correctly. This is done by having the tasks run for their argument's number of milliseconds. A round robin task is created that toggles channel 2 every 3

milliseconds. A periodic task is created that turns on channel 2 and lasts 5 milliseconds, with a start delay of 3 ticks (15ms). Finally a system task is created at the beginning that turns on channel 0 and runs for 7 milliseconds before turning off channel 0 and destroying itself.

Figure 9: Task Arguments Test Results

7.2.7 Services Test

The services test consists of one system task that sets port 1 on and subscribes to a service. When the service is published to, the task resumes and turns port 1 off for 1 ms before re-subscribing. A round robin task controlling channel 2 does the exact same behavior. Finally 2 round robins exist that simply turn channel 3 or channel 4 on respectively when they are running. A periodic task is used to publish values to the two subscribing tasks and controls channel 0. The periodic task waits 1ms after publishing before it relinquishes control. This displays task preemption, and both round robin and system task subscription.

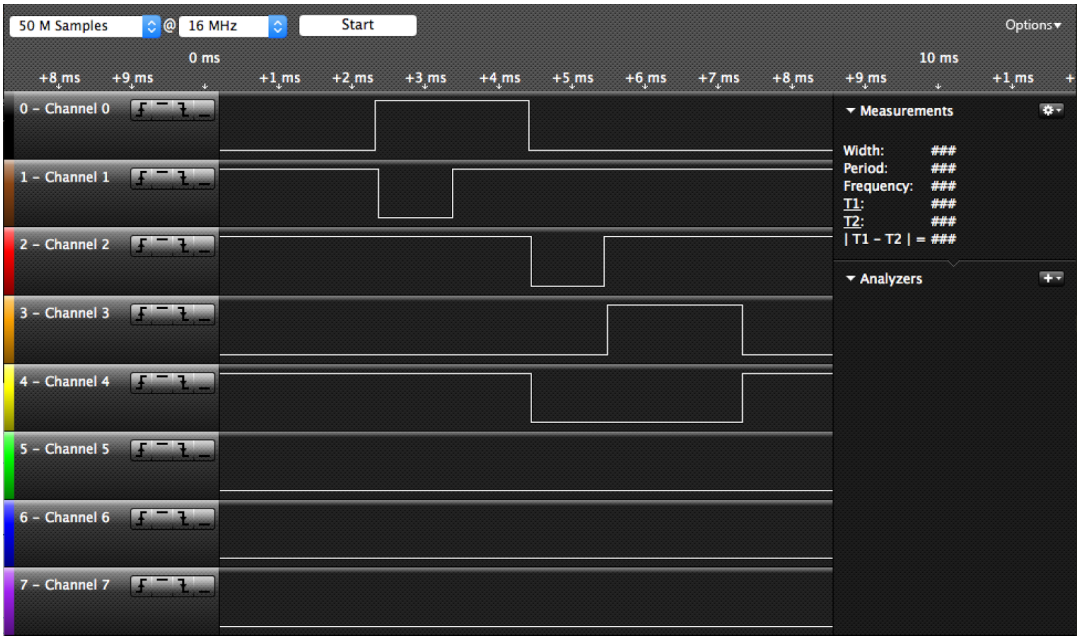

Figure 10: Services Test Results

7.2.8 Service Values Test

The service values test consists of 3 round robin tasks controlling channel 1, 2 and 3 respectively. When a round robin task is running, its channel is set on and the other two are set off. The first two round robin tasks subscribe to a service after 5ms of running. When the service is published to, the first task will resume if the published value is 0, and the second will resume if the published value is 1. A periodic task runs that alternates publishing 0 and 1 with a period of 6 ticks.

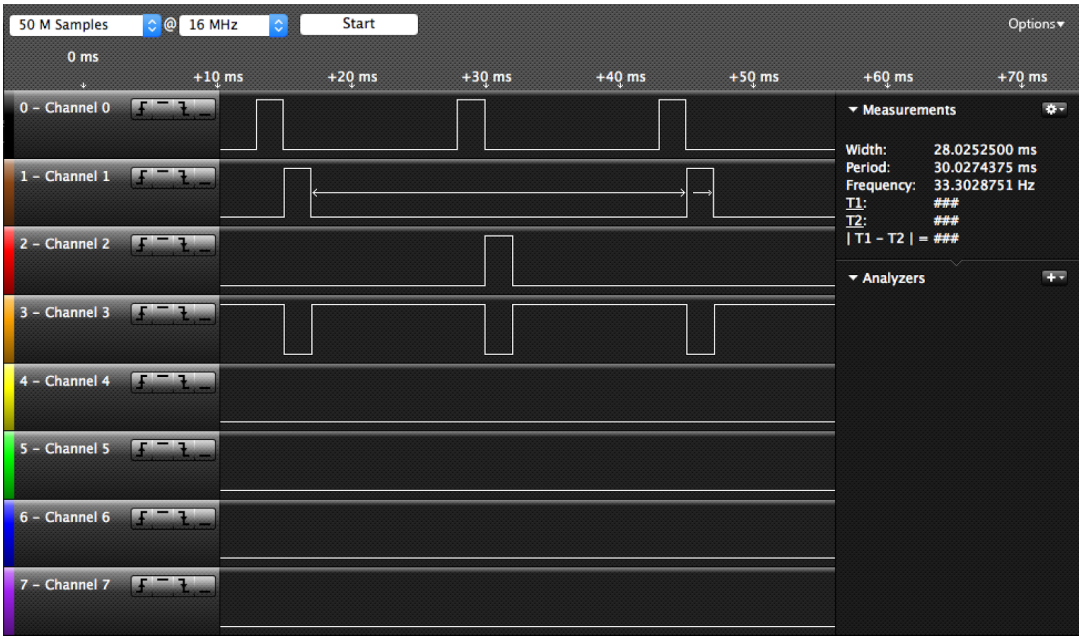

Figure 11: Service Values Test Results

7.2.9 Task Freeing Test

The task freeing test is designed to ensure that task memory is reallocated whenever a task is terminated. To do this a set of 6 tasks are created, 3 system and 3 round robin that control channels 0 through 5 in order (System first). Each task spawns, sets its corresponding channel on, busy waits 1ms then sets the channel off before terminating. A periodic task runs every 10 ticks that creates 6 new tasks with the exact same purpose. If the system works correctly, no errors should be thrown, and infinite tasks should be able to be created (7 concurrently).

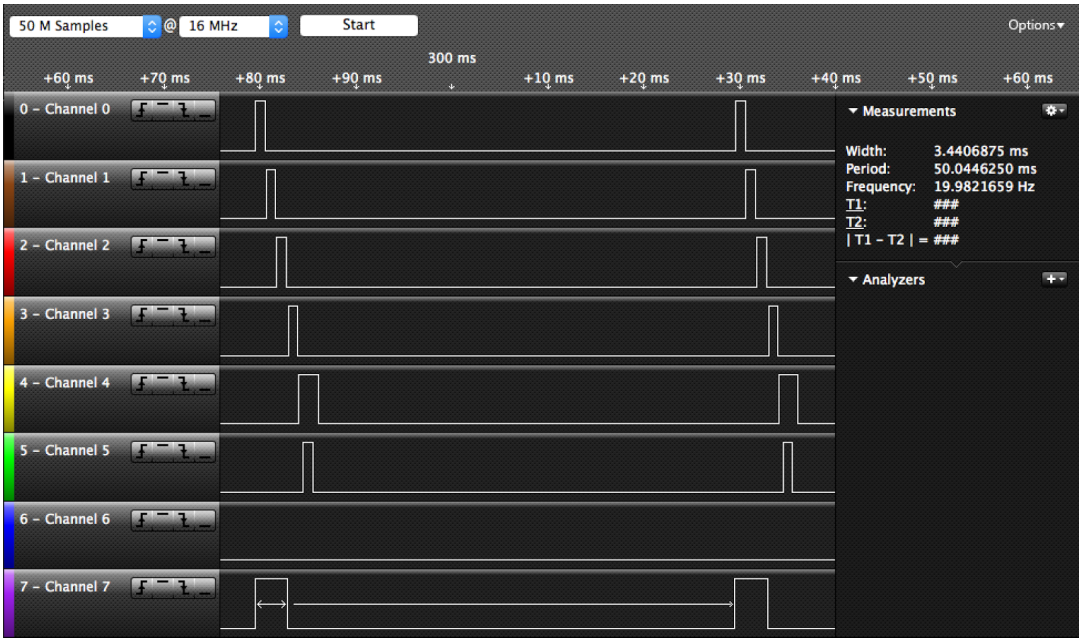

Figure 12: Task Freeing Test Results

7.3 Error Test Cases

Error test cases attempt to force errors to occur in order to prove that the RTOS's error detection is correctly implemented. All error test cases are tested by running on the board, and observing the displayed error code using the board's built in LED. Certain errors cannot actually occur within the RTOS but exist more as error correction and bug detection methods. These errors are not tested using unit tests.

7.3.1 WCET Too Long

The WCET error is tested by simply creating a periodic task with a longer WCET than period. The system immediately errors out and displays the correct error code.

7.3.2 Max Services

The max services test is also trivial. The test system attempts to create 9 services (1 more than the predefined max). When the 9th service is allocated, the RTOS aborts and prints the corresponding error code.

7.3.3 User OS Abort

The os abort test is done by having a user create task call OS_Abort. When this happens, the system error message cannot be set, so this error acts as a default. If OS_Abort is called in user generated code, the correct error code is displayed.

7.3.4 Max Tasks Allocated

The max tasks test is used to ensure that if the user attempts to create more tasks than the OS supports, the system will abort. The OS supports 8 tasks, but one is reserved for the RTOS's idle task. The test attempts to create 8 tasks, breaking the limit by one. One the 8th task is created, the system aborts and the correct error message is displayed.

7.3.5 Periodic Runs Too Long

The periodic error task is fairly simple. A periodic task is generated with a preset WCET. The task enters an infinite loop, which causes it to run longer than it's WCET. In this case the operating system aborts after the WCET has passed, and the correct error code is displayed.

7.3.6 Periodic Collision

The periodic collision test is done by creating two periodic tasks, the first of which runs for 6 ticks. The second task attempts to start at 3 ticks. This causes a collision, and the correct error code is displayed by the system.

7.3.7 Periodic Subscription Attempt

The periodic subscription test is also simple. A single periodic task is created. That periodic task immediately attempts to subscribe to a service. When this occurs, the OS aborts and displays the correct error code.

8.0 Conclusion

By porting an existing RTOS over to the ATmega2560, and adapting it to our needs, we were able to create our own RTOS for use in creating more advanced programs to run on the board. Handling context switching, timing, task management, and subscription based inter-task communication, a flexible, fast, and easily usable system was created.

A number of unit tests were also created to test the major features of the RTOS. By creating these tests, we were able to ensure the OS worked as intended, and was capable of detecting user errors before they broke the system itself.

Some optimizations and timing issues do exist within the system. These are minor issues that should not affect the vast majority of use cases, however it is possible to do some odd things such as giving a periodic task an infinite WCET. Outside of some potential abuses, the system is well tested, and works as expected.

Our resulting code is available on Github [4].

9.0 References

[1] http://www.obdev.at/products/crosspack/download.html
[2] https://webhome.csc.uvic.ca/~mcheng/460/summer.2013/samples/krazanowski_a2.pdf
[3] https://www.saleae.com/downloads
[4] https://github.com/Gomer3261/CSC460