ECE 121
Lab 2: Rotary Encoders, Ping Sensors, and R/C servos

TIMERS, SPI, INPUT CAPTURES, AND OUTPUT CAPTURES

MEL HO
STUDENT ID #: 1285020
WINTER 2021

February 11, 2021

1 Introduction and Overview

Lab 2 familiarizes us with how to use a ping sensor and absolute rotary encoder to control an R/C servo motor. Through this process, we learn how to use timers, the **Serial Peripheral Interface** (SPI), **Input Captures**, and **Output Captures** on our microcontroller. Once the lab project is fully built, we connect to our lab interface to communicate with our UNO32 to set which control input to use and display the current servo angle between ±60°. Given that lab 2 cannot be fully captured by simply using screenshots, Appendix B. provides a google drive link to video clips taken to showcase the lab 2 interface functioning as intended.

2 Timers

The first part of this lab we tackled is the time events on the UNO32 microcontroller. For this section, we designed a free running timer using the 16-bit "B" timer, timer 5 which rolls over every millisecond. This is done by first initializing timer 5 on the UNO32 and setting the clock rate to be 1KHz.

```c
void FreeRunningTimer_Init(void){
    T5CON = 0; // disable timer 5
    TMR5 = 0; // reset timer 5
    PR5 = FPB / TIMERFREQ / 4; // number of ticks for 1KHz
    T5CONbits.TCKPS = 0b010; // set pre-scaler to 1:4

    IPC5bits.T5IP = 3; // Sets priority to 3
    IPC5bits.T5IS = 1; // Sub-priority to 1

    IFS0bits.T5IF = 0; // Sets the interrupt flag to 0
    IEC0bits.T5IE = 1; // Turns on timer interrupt

    T5CONbits.ON = 1; // enables timer 5
}
```

To calculate the value for our **period rate**, the following equation used:

$$\mathrm{PR} = F_{PB}/F_{\mathrm{desired}}/\mu \tag{1}$$

where F_{PB} is our Peripheral Bus Frequency, F_{desired} is the frequency we want our timer to roll over, and μ is our prescaler value which we use to fit our value into a 16-bit timer. In this context our $F_{\mathrm{desired}} = 1000$ and a prescaler of $\mu = 4$ was chose to fit our total tick value nicely into $2^{16} = 65536$. With this configuration, our timer 5 is set to roll over every millisecond. Next, the interrupt is coded such that two static int variables are incremented every time the interrupt is triggered:

```c
void __ISR(_TIMER_5_VECTOR, ipl3auto) Timer5IntHandler(void)
{
    IFS0bits.T5IF = 0;
    milliseconds++;
    microseconds += 1000;
}
```

Once implemented, a test harness was created to do the following:

- Print out a debug message of the date and time the program was compiled
- Turn on LEDs on the I/O shield every 2 seconds
- Toggle an output pin on and off at a frequency of 0.5Hz
- Print out the number of milliseconds and microseconds that have passed since the program started every 2 seconds

With our free running timer functional, the next step in lab 2 is interfacing with our absolute rotary encoder.

3 Rotary Encoder

In order to use our rotary encoder as a control input, we must first set up communication between it and the microcontroller. This is done through setting up the serial peripheral interface on our UNO32 and sending specific communication packets to the encoder.

3.1 Serial Peripheral Interface

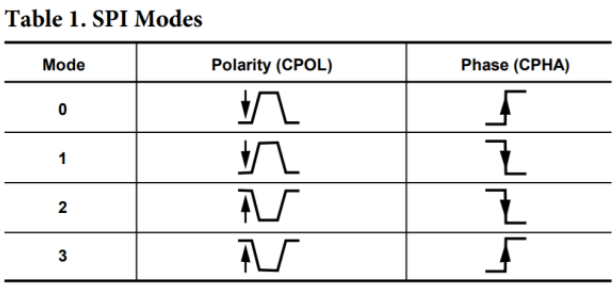


Table 1. SPI Modes

Mode	Polarity (CPOL)	Phase (CPHA)
0		
1		
2		
3		

Figure 1: Different modes of SPI.

The serial peripheral interface was set up to have a 5Hz clock speed, be in SPI mode 1, 16-bit mode, and have the PIC32 as our master; this is done by using the following initialization:

```
int RotaryEncoder_Init(char interfaceMode)
{
    if (interfaceMode == ENCODER_BLOCKING_MODE)
    {
        SPI2CON = 0; // disable SPI
        SPI2CONbits.MSTEN = 1; //set to MASTER mode
        SPI2CONbits.SMP = 1; // Input is sampled on the end of the clock signal
        SPI2CONbits.CKP = 0; // Idle state is low
        SPI2CONbits.CKE = 0; // data shifted out/in from idle to active
        SPI2CONbits.MODE32 = 0; // 32-bit mode is disabled
        SPI2CONbits.MODE16 = 1; // 16-bit mode is enabled
        SPI2BRG = (FPB / DESIRED_FREQ / 2) - 1; // set to 5Hz
        SPI2BUF; // Clears SPI buffer


        SS_PIN_10_INIT; // initialize CS, pin 10

        SPI2CONbits.ON = 1; // enables SPI
    }
    return SUCCESS;
}
```

As shown in Fig. 1, SPI can be configured into different modes depending on what we component we want to communicate with. Since our encoder requires us to be in mode 1 for communication, we have set our init to accomodate for that. Calculating our **Baudrate Generator**(SPIBRG), which determines our frequency, is done use the equation provided by the FRM below:

$$\text{SPIxBRG} = \frac{F_{PB}}{2 \cdot F_{\text{SCLK}}} - 1 \tag{2}$$

This gives us a value of 3. Pin 10 was arbitrarily picked as our slave pin that will be used in our SPI transmission. For human-readability, a macro was used. Please consult appendix A for code details. Now that the SPI register has been set up, we will implement the specific SPI transactions necessary to communicate with our AMS rotary encoder.

3.2 Absolute Encoder

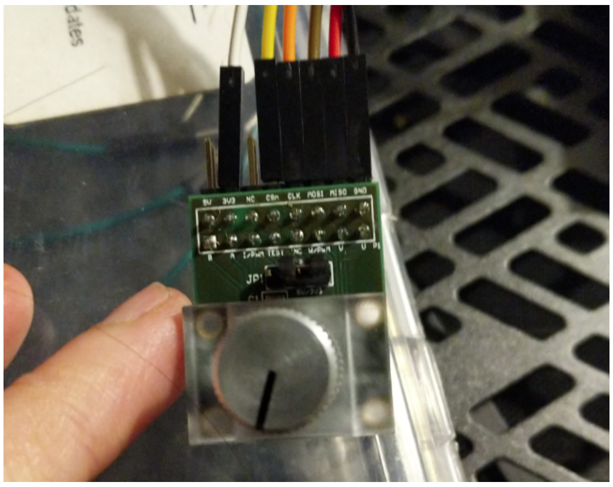


Figure 2: AMS Absolute Rotary Encoder.

Based on the AMS rotary encoder documentation, the encoder has two different frames: the command and data frame. For our desired application, we will be sending a command frame over in order to receive a data frame of the raw angles that the encoder is measuring. Our command frame is composed of the following bits. `[parity_bit R/W_bit Address (bits 13:0)]`. The **parity bit** attached must be *even*, otherwise the command will be rejected by the AMS encoder and we can set whether we are sending a **read** (1) or **write** (0) command.

3.2.1 Parity Check

```
Algorithm 1: Function to check parity
Input: $in$, integer to check parity
Result: $p$, parity bit, 0 or 1

$p \leftarrow 0$ ;

for $i \leftarrow 1$ to 0x8000 do
    if ($in$ & $i$) then
        $p \leftarrow p + 1$
    end
    $i \leftarrow (i \ll 1)$ ;
end
$p \leftarrow (p\%2)$ ;
return ($p$) ;

Insert Parity Bit into Payload: $in \leftarrow in|(p \ll 15)$ ;
```

Figure 3: Pseudo-code for parity check function. Taken from Lab 2 manual.

The parity bit is used to check whether the bits are odd or even. Based on the pseudo code in Fig. 8, we can implement the following to commute our parity bit of our message:

```
  unsigned short checkParity(unsigned int in)
{
    unsigned short parity = 0;
    int i;
    for(i=0x0001; i <= 0x8000; i <<= 1)
    {
        if (in & i)
            parity += 1;
    }
    parity = parity % 2;
    return parity;
}

```

our function increments through each bit in our command message and increments the parity counter if `i == in`. Then it determines whether the parity is 0 or 1 by `parity = parity % 2`. Once the parity has been calculated, we add it to our message by shifting the value to bit 15. (`parity << 15`). We will see an example of this when performing our first command message, `DIAAGC`.

3.2.2 AMS Diagnostic Message

To successfully send a SPI transmission to the encoder and read a response, we need to do the following steps:

1. **CS** is pulled to low
2. delay by $1\mu s$ (delay minimum is $350ns$)
3. Send command to **SPI2BUF**
4. Wait until buffer is empty
5. **CS** is pulled to high
6. delay by $1\mu s$
7. discard value in buffer.
8. **CS** is pulled to low
9. delay by $1\mu s$
10. Send **NOP** command to **SPI2BUF**
11. Wait until buffer is empty
12. **CS** is pulled to high
13. delay by $1\mu s$
14. store value from **SPI2BUF**

The first read value inside SPI2BUF is discarded because the actual data is sent after the initial handshake. To test this sequence, we sent of a `DIAAGC` command to the rotary encoder using the `0x3FFC` address associated. This address then has a write bit and parity added and sent following the sequence listed above.

```
#define NOP ((1 << 14) + 0x0000)
#define SEND_NOP (NOP | (checkParity(NOP) << 15))
#define DIAAGC ((1 << 14) + 0x3FFC)
#define SEND_DIAAGC (DIAAGC | checkParity(DIAAGC) << 15)

SS_LOW;
delay_us(1);
SPI2BUF = SEND_DIAAGC;
while(!SPI2STATbits.SPIRBF);
```

```
SS_HIGH;
delay_us(1);
SPI2BUF;

SS_LOW;
delay_us(1);
SPI2BUF = SEND_NOP;
while(!SPI2STATbits.SPIRBF);
SS_HIGH;
delay_us(1);

DIAAGC = (0x3FFF & SPI2BUF); // store DIAAGC value with 15:14 masked off)

```

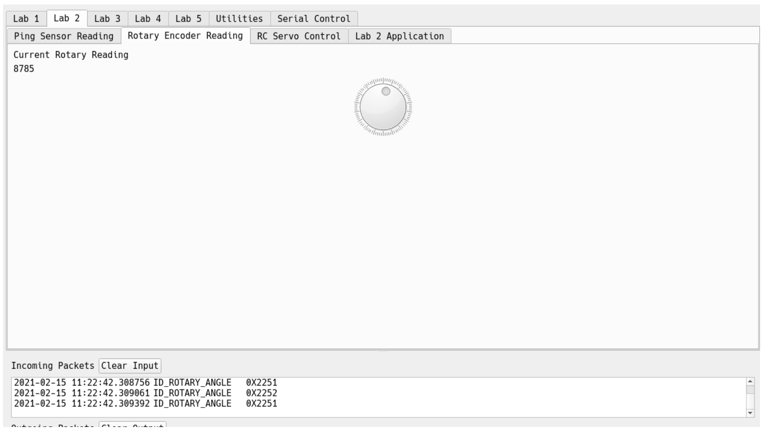
After successfully sending and receiving a `DIAAGC` message from our rotary encoder, it's time to get the raw angles.

3.2.3 AMS Raw Angle

Receiving raw angles from the AMS encoder uses the same code sequence as the `DIAAGC` command, except using the `ANGLECOM` address: `0x3FFF`. We store this transaction in the RotaryEncoder function, `ReadRawAngle()`.

```
  unsigned short RotaryEncoder_ReadRawAngle(void)
{
    SS_LOW;
    delay_us(1);
    SPI2BUF = READ_ANGLECOM;
    while(!SPI2STATbits.SPIRBF);
    SS_HIGH;
    delay_us(1);
    SPI2BUF;

    SS_LOW;
    delay_us(1);
    SPI2BUF = SEND_NOP;
    while(!SPI2STATbits.SPIRBF);
    SS_HIGH;
    delay_us(1);

    rawAngle = (0x3FFF & SPI2BUF);
    return(rawAngle);
}

```

Using a test harness, we successfully can display raw angles to our Lab Interface as shown in Fig. 4.



Current Rotary Reading
8785

```
2021-02-15 11:22:42.308756 ID_ROTARY_ANGLE   0X2251
2021-02-15 11:22:42.309061 ID_ROTARY_ANGLE   0X2252
2021-02-15 11:22:42.309392 ID_ROTARY_ANGLE   0X2251
```

Figure 4: Lab 2 Interface rotary encoder position.

3.3 Ping Sensor

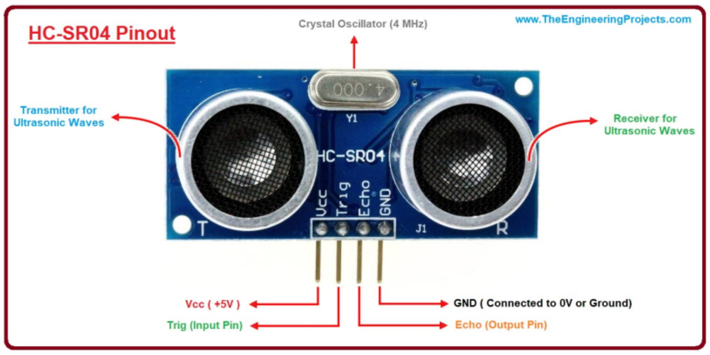

HC-SR04 Pinout
Crystal Oscillator (4 MHz)
www.TheEngineeringProjects.com
Transmitter for Ultrasonic Waves
Receiver for Ultrasonic Waves
HC-SR04
Vcc Trig Echo GND
T
R
Y1
J1
Vcc (+5V)
Trig (Input Pin)
GND (Connected to 0V or Ground)
Echo (Output Pin)

Figure 5: HC-SRO4 Ultrasonic Ranging Sensor. Taken from Lab 2 manual.

As another control input for our R/C servo, we utilize a time-of-flight sensor commonly known as a "ping" sensor. This sensor uses ultrasound to measure the distance (cm) from an object based on the reflection of its transmitted signal.

3.3.1 Initialization and Input Capture

In order to properly interface with our HC-SRO4 ping sensor, we must configure our UNO32 to have:

- Timer4 with a 60ms pulse

- a trigger pin which needs to be high for at least $10\mu s$
- an echo pin that uses an **input capture** interrupt

The input capture subsystem is used to accurately measure the pulse width generated by our ping sensor. It copies the clock of the timer its synced with to a capture register whenever an external event occurs. To initialize this, the following was implemented:

```
int PingSensor_Init(void)
{
    T4CON = 0;
    TMR4 = 0;

    T2CON = 0;
    TMR2 = 0;

    IC3CON = 0;
    IC3CONbits.ICTMR = TIMER_2; //TIMER2
    IC3CONbits.ICM = SIMPLE_CAPTURE_EVERY; // every edge
    IC3CONbits.FEDGE = FIRST_CAPTURE_RISE;
    IC3CONbits.ICI = INTERRUPT_EVERY_CAPTURE;
    IC3BUF; // clear input capture buffer

    IPC3bits.IC3IP = 2; //IC3 PRIORITY (higher than timer 2)
    IFS0bits.IC3IF = 0;
    IEC0bits.IC3IE = 1;

    T2CONbits.TCKPS = 0b100; // 1:16
    T4CONbits.TCKPS = 0b110; // 1:64

    // I set both PRs to my desired tick value, given the frequency I want that is then prescaled to fit a 16-bit
      timer.
    PR2 = FPB / TIMERFREQ_TR2 / PRESCALER_16; // interrupt every 20ms / 20000us
    PR4 = FPB / TIMERFREQ_TR4 / PRESCALER_64; // interrupt at 60ms (16.666667Hz)

    IPC4bits.T4IP = 1; // TIMER 4 PRIORITY
    IPC4bits.T4IS = 1;

    IPC2bits.T2IP = 3; // TIMER 2 PRIORITY
    IPC2bits.T2IS = 2;

    IFS0bits.T4IF = 0;
    IEC0bits.T4IE = 1;

    TRIGGER_PIN_INIT;
    TRIGGER_PIN = LOW;

    T4CONbits.ON = TRUE;
    T2CONbits.ON = TRUE;
    IC3CONbits.ON = TRUE;

    return SUCCESS;
}

```

We first configure **Timer4** which will be our timer that rolls over every $60ms$. Then we set our Input Capture (**IC3**) to sync with **Timer2** which rolls over every $20ms$. IC3 is then synced to Timer2, interrupts on every capture, captures on every edge, and starts initially as a rising edge. Finally, a **trigger pin** is designated and set to as an output; it will be used to trigger the ping sensor's ultrasonic pulse which will be measured by our IC3 pin.

3.3.2 Trigger

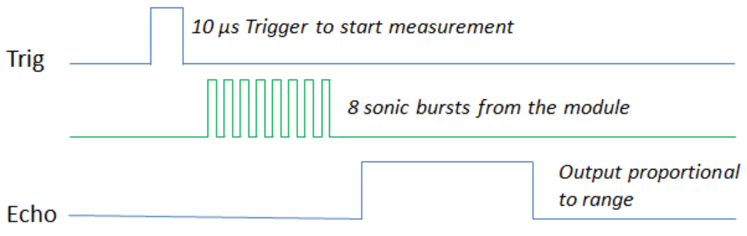

Trig
10 µs Trigger to start measurement
8 sonic bursts from the module
Echo
Output proportional to range

Figure 6: Ping sensor sequence.

For triggering our ping sensor, we will be using the Timer4 interrupt.

```
void __ISR(_TIMER_4_VECTOR) Timer4IntHandler(void)
{
    IFS0bits.T4IF = 0;
    TRIGGER_PIN = HIGH;
    delay_us(10);
    TRIGGER_PIN = LOW;
}

```

Every 60ms, our trigger pin is set to high for 10μs and then toggled low. This triggers eight sonic bursts from the HC-SRO4 that reflect back from the object and is measured by the echo, as seen in Fig. 6.

3.3.3 Echo

We handle the time difference measured by our echo inside our input capture interrupt. This is done with a switch statement that compares the expected edge state with the measured state from our IC3 pin. If it the values are correct, they will be stored in their appropriate variables and the timer interval of the current distance will be calculated. The **IC3BUF** is constantly cleared during this process to prevent a halt due to overflow.

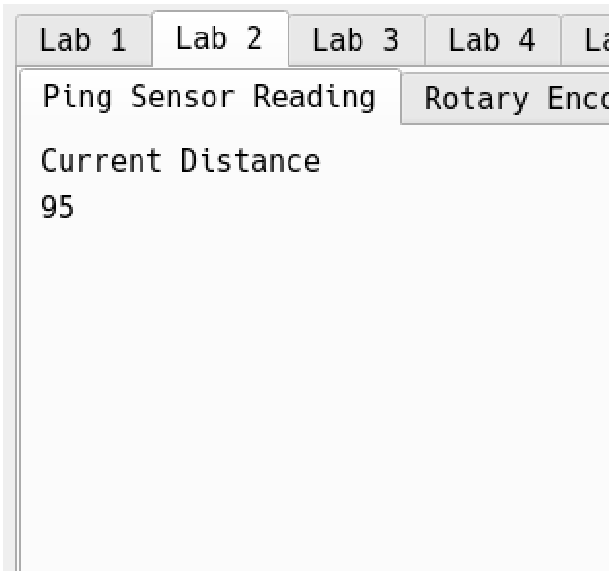
```
void __ISR(_INPUT_CAPTURE_3_VECTOR) __IC3Interrupt(void)
{
    IFS0bits.IC3IF = 0;
    switch(edgeState)
    {
        case RISING_EDGE:
            if (edgeState == PORTDbits.RD10)
            {
                uptime = (0xFFFF & IC3BUF);
                edgeState = FALLING_EDGE;

                // Clear buffer if it's still not empty
                IC3BUF;
            }
            break;

        case FALLING_EDGE:
            if (edgeState == PORTDbits.RD10)
            {
                downtime = (0xFFFF & IC3BUF);
                timeInterval = downtime - uptime;
                edgeState = RISING_EDGE;

                // Clear buffer if it's still not empty
                IC3BUF;
            }
            break;
    }
            IC3BUF;

}
```

3.3.4 Distance Calculation

Now that we have a consistent and accurate time interval, the distance of an object from the ping sensor can be measured using the following equation:

$$d = \frac{\Delta t \cdot 4\mu}{2 \cdot F_{\text{desired}}} \cdot \frac{1}{58} \tag{3}$$

where μ is our pre-scaler value of Timer2 (1:16), F_{desired} is our current Timer2 frequency (50Hz), and Δt is our time interval. This equation first converts our tick values into μs and then into centimeters using the conversion rate, $1\mu s/58$ found in our ping sensor spec sheet. Testing our functions, the ping sensor was able to send accurate distance values in centimeters to our lab interface.



Current Distance
95

Figure 7: Lab 2 Interface ping sensor distance in centimeters.

3.4 R/C Servo Motor

Figure 8: A R/C servo. Taken from the Lab 2 manual.

The final library necessary for our lab 2 interface is the R/C servo motor driver. For this part, we will be using an output capture to generate precise pulses that drive our servo to specific angles within ±60°. Like the input capture, the output capture will be synchronized to a timer register.

3.4.1 Initialization

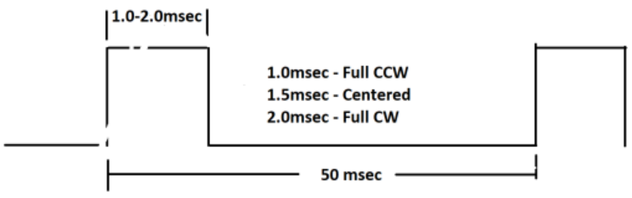


Figure 9: Pulse Width Command for RC Servo. Taken from Lab 2 manual.

As shown in our init code, we configure out timer. As specified in the Lab 2 manual, our **Timer3** will rollover with a frequency of 20Hz. Our output capture (**OC3**) is turned on and synchronised to Timer3. Fig. 9 shows the pulse width range of our R/C servo; our R/C motor is controlled by microseconds ticks ranging from $600\mu - 2400\mu$. Before we set our $OCRS$ register that determines the position of our R/C servo, we will need to convert our motor tick commands into millisecond tick values equivalent to our timer's rollover rate. The conversion is calculated using the following equation:

$$t = \frac{t_{\text{total}}}{T} \cdot (\frac{\text{pulse}}{1000}) \tag{4}$$

where t is the corresponding motor tick value, t_{total} is the number of ticks for 20Hz, T is the millisecond period, pulse is our desired microsecond pulse value converted into milliseconds. On startup, our R/C servo's position is centered (1500μs) and the conversion is calculated during pre-processing.

```c
#define TIMER_3_PULSE FPB / TIMEFREQ_TR3 / TIMER_PRESCALE_32 // 62500 ticks

#define MS_PERIOD 50 // 50ms

#define CENTER_PULSE (TIMER_3_PULSE / MS_PERIOD) * (RC_SERVO_CENTER_PULSE * .001)

#define MIN_PULSE (TIMER_3_PULSE / MS_PERIOD) * (RC_SERVO_MIN_PULSE * .001)

#define MAX_PULSE (TIMER_3_PULSE / MS_PERIOD) * (RC_SERVO_MAX_PULSE * .001)

int RCServo_Init(void)
{
    T3CON = 0;
    TMR3 = 0;

    T3CONbits.TCKPS = 0b101; // 1:32 prescale
    PR3 = TIMER_3_PULSE;

    IPC3bits.T3IP = 3; // Timer 3 priority
    IPC3bits.T3IS = 2;

    OC3CON = 0;
    OC3CONbits.OCTSEL = TIMER_3;
    OC3CONbits.OCM = 0b101;
    OC3RS = CENTER_PULSE; //1500us default orientation is center

    IFS0bits.OC3IF = 0; //OC3 priority
    IEC0bits.OC3IE = 1;

    T3CONbits.TON = TRUE;
    OC3CONbits.ON = TRUE;
    return SUCCESS;
}
```

3.4.2 Setting Pulse

With our output capture set up, a `setPulse()` function was implemented to change the angle of our R/C servo and keep it within the range of acceptable motor ticks.

```c
int RCServo_SetPulse(unsigned int inPulse)
{
    if (inPulse > RC_SERVO_MAX_PULSE || inPulse < RC_SERVO_MIN_PULSE)
        return ERROR;

    if (inPulse != prevPulse)
    {
        prevPulse = inPulse;

        // conversion from timer ticks to microseconds
        OC3RS = (TIMER_3_PULSE / MS_PERIOD) * (inPulse * 0.001);
    }


    return SUCCESS;
}
```

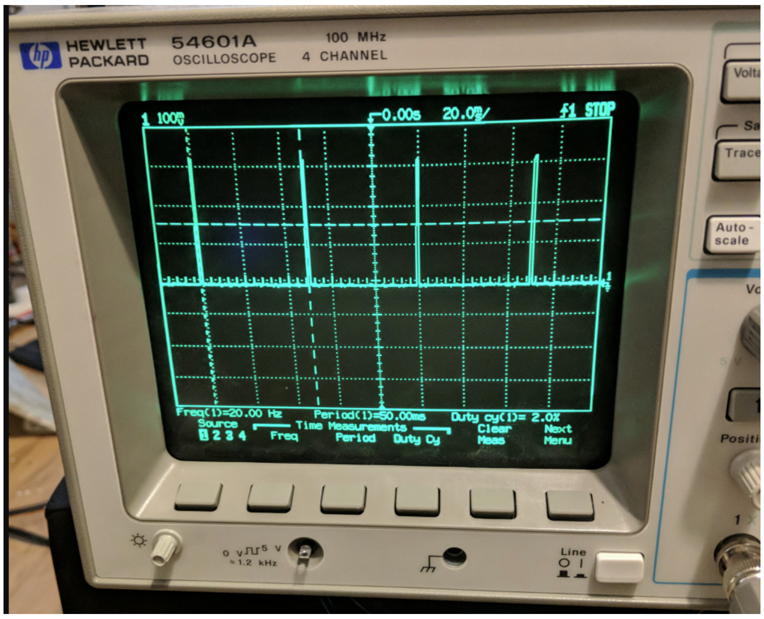


Figure 10: Oscilliscope trace of our 20Hz timer with a pulse-width of 1500μs.

If our new pulse command is different from the current pulse command and it is within our motor tick range, we set our **OC3RS** to the new pulse value. See Appendix A for the rest of the helper functions available. With the R/C motor driver implemented, the setting function is tested using different motor tick values within and out of range. Fig. 10 shows an oscilloscope trace of a 1500μs pulse triggered ever 50ms. Connecting the R/C motor to this output signal, produced the expected angles.

4 Lab 2 Interface

With all necessary libraries implemented, a main program was created to interact with the Lab 2 interface. This main program would control the R/C servo motor in either the ping sensor mode or rotary encoder mode based on the `ID_LAB2_INPUT_SELECT` ID. When in the rotary encoder control mode, the angle values are sent using the `LAB2_ANGLE_REPORT` ID; this packet contains a union of a signed int that holds the angle in millidegrees and an unsigned char which represents whether the angle is within the motor tick range or not every 50ms. `0x02` is if the angle is within bounds, `0x04` is if the angle is negative out of bounds, and `0x01` is if the angle is positive out of bounds. This union is expressed by the following code block:

```
union AngleData {
    struct {
```

```c
        signed int Angle;
        unsigned char status;
    };
    char asChar[5];
};
```

For each control mode, we need to convert our original measured values into the correct motor ticks. Since our ping sensor is measured in centimeters, the following conversions were used:

```c
case(PING_SENSOR):

  // Get Ping Value in cm
  pingValue = PingSensor_GetDistance();

  // checks to see if ping value is between 25cm-125cm
  if (pingValue >= 25 && pingValue <= 125)
  {
    // Converts cm to Angles (from lab 2 for += 60 deg)
    cmToAngle = -60 + (120 * ((pingValue * 0.01) - 0.25));

    // Scaling angles to tick values
    angleToTicks = 1500 + (cmToAngle * 15);

    // binds ping values to max and min pulse rates
    if (angleToTicks > RC_SERVO_MAX_PULSE)
        angleToTicks = RC_SERVO_MAX_PULSE;

    if (angleToTicks < RC_SERVO_MIN_PULSE)
        angleToTicks = RC_SERVO_MIN_PULSE;

    RCServo_SetPulse(angleToTicks);

  }
    break;
```

We first convert our centimeters into angles using the equation given in the lab 2 manual. Then, these angles are mapped to tick values accordingly. Testing our ping sensor mode with our motor, the servo rotates in accordance to the distance of our target to the the HC-SR04. Please consult the video clip provided in Appendix B. for more details. Our encoder code is slightly more complicated as we need to not only convert our raw angles into motor ticks, but also send our angles in millidegrees to the lab 2 interface.

```c
case(ENCODER):

    rawAngle = RotaryEncoder_ReadRawAngle();

    // 0-360 deg conversion
    //Converts raw angle ticks to degrees. 360 / 16200 (largest deg/tick size) = 0.02222
    angleConversion = (rawAngle * .022);

    // converts from 0-360 deg => +-180 deg
    if (angleConversion > 179)
        angleConversion = angleConversion - 360;

    // Linear interpolation is used to map the converted angles to values that fit the range of 600-2400us
    // normalization: (angle + (angle_difference / 2 )) / angle_difference. In this case angle_diff = 120 deg
    lerpRatio = (angleConversion + (ANGLE_DIFF / 2)) * ANGLE_DIFF_FRACTIONAL;

    // returns mapped values into motor us
    mappedAngle = lerp(RC_SERVO_MIN_PULSE,RC_SERVO_MAX_PULSE,lerpRatio);

    // Converts degrees to millidegrees and Big Endian
    angleReport.Angle = Protocol_IntEndednessConversion((signed int)(angleConversion * 1000));

    // Checks to see if the encoder angle is between the RC-SERVO range (+-60 deg)
    if (mappedAngle >= RC_SERVO_MIN_PULSE && mappedAngle <= RC_SERVO_MAX_PULSE)
    {
        angleReport.status = IN_RANGE;
        RCServo_SetPulse(mappedAngle);
    }
```

```
29      else if (mappedAngle < RC_SERVO_MIN_PULSE)
30          angleReport.status = UNDER_RANGE;
31
32      else if (mappedAngle > RC_SERVO_MAX_PULSE)
33          angleReport.status = OVER_RANGE;
34
35      break;
```

We first grab our current raw angle and then convert this step value into degrees ranging from $0° - 360°$. Then, we shift our angles to represent a range of $\pm 180°$. This range is then mapped to motor tick values using **linear interpolation**. Linear interpolation is a curve fitting method which constructs new data points given a range of known data points. Since we know the min and max motor tick values and they are linear, we can use that information to calculate a ratio for $\pm 60°$ as shown in line 15 of our code block. The linear interpolation function is shown below:

```
/**
 * @Function lerp(uint min, uint max, float ratio)
 * @param min, max, ratio
 * @return uint
 * @brief Linear interpolation function used to map our encoder angles into a range between 600-2400 motor ticks.*/
unsigned int lerp(unsigned int min, unsigned int max, float ratio)
{
    return min + ratio * (max - min);
}
```

This function takes in our minimum and maximum motor tick values, and a ratio calculated from our current angle and maps it to a discrete value within that range. After this conversion, depending on if the motor tick value is within our operating range, we set our R/C servo to the new pulse value and set the status accordingly. Finally we convert take our angle in degrees and convert it to big endian and millidegrees and send an angle report every $50ms$ using our free running timer.

```
milliseconds = FreeRunningTimer_GetMilliSeconds();
if (milliseconds >= desired_ms)
{
    desired_ms += 100;
    Protocol_SendMessage(5, ID_LAB2_ANGLE_REPORT, &angleReport);
}
```

Please see Appendix B. for video footage demoing this main program.

5 Conclusion

In this lab, we learned how to use the timer, SPI, input capture, and output subsystems to create a program that controls a R/C servo motor using either a rotary encoder and ping sensor. This lab proved to be challenging in the sense that it required enough timer precision such that each physical peripheral attached could work as expected; the SPI subsystem was the most challenging out of the four components as diagnostic was hard due to a lack of sufficient measuring tools and information provided by the AMS encoder. After multiple days of troubleshooting, the problem ended up being a hardware issue where a jumper needed to be shifted in order to have the correct angles displayed. The magnet was also incorrectly assembled and had to be shifted for the knob to accurately match the lab 2 interface's GUI. Despite these setbacks, a deeper understanding about clock rate ticks and how each subsystem works in conjunction with timers was gained; this will hopefully help expedite overcoming these kind of challenges in future labs.

Appendix

A. Source Code

5.0.1 Lab 2 Application

```c
/*
 * File:   lab2Application.c
 * Author: Mel Ho
 *
 * Created on February 8, 2021, 10:54 AM
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include "xc.h"
#include "BOARD.h"
#include "stdio.h"
#include "Protocol.h"
#include "FreeRunningTimer.h"
#include "MessageIDs.h"
#include "RCServo.h"
#include "RotaryEncoder.h"
#include "string.h"
#include "stdio.h"
#include "delays.h"
#include "PingSensor.h"
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define PING_SENSOR 0
#define ENCODER 1
#define ANGLE_DIFF 120 // 120 deg
#define ANGLE_DIFF_FRACTIONAL 0.00833 // 1 / 120
#define IN_RANGE 0x02
#define UNDER_RANGE 0x04
#define OVER_RANGE 0x01
static char message[MAXPAYLOADLENGTH];
static unsigned short rawAngle = 0;
static signed short angleConversion = 0;
static unsigned int mappedAngle = 0;
static float lerpRatio = 0; //linear interpolation ratio
static unsigned short pingValue = 0;
static signed short cmToAngle = 0;
static unsigned short angleToTicks = 0;
static unsigned char ID;
static unsigned int payload;
static unsigned int milliseconds;
static unsigned int desired_ms = 50;
union AngleData {
    struct {
        signed int Angle;
        unsigned char status;
    };
    char asChar[5];
};
/*******************************************************************************
 * PUBLIC FUNCTIONS                                                           *
 ******************************************************************************/
/**
 * @Function lerp(uint min, uint max, float ratio)
 * @param min, max, ratio
 * @return uint
 * @brief Linear interpolation function used to map our encoder angles into a range between 600-2400 motor ticks.*/
unsigned int lerp(unsigned int min, unsigned int max, float ratio)
{
    return min + ratio * (max - min);
}
/*******************************************************************************
 * MAIN  PROGRAM                                                               *
 ******************************************************************************/
int main(void)
{
    BOARD_Init();
    FreeRunningTimer_Init();
    RCServo_Init();
    PingSensor_Init();
    Protocol_Init();
    RotaryEncoder_Init(ENCODER_BLOCKING_MODE);
    // Send ID_DEBUG Message of Compilation Date
    sprintf(message, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
    // Holds Control Mode
    unsigned int controlMode = ENCODER;
    while(1)
    {
        ID = Protocol_ReadNextID();
        Protocol_GetPayload(&payload);
        union AngleData angleReport; // angle report for the encoder
        // Switch between the inputs (encoder/ping) based on the ID_LAB2_INPUT_SELECT message
        if (ID == ID_LAB2_INPUT_SELECT)
            controlMode = payload;
        switch(controlMode)
        {
            case(PING_SENSOR):
              // Get Ping Value in cm
              pingValue = PingSensor_GetDistance();
              // checks to see if ping value is between 25cm-125cm
              if (pingValue >= 25 && pingValue <= 125)
              {
                // Converts cm to Angles (from lab 2 for += 60 deg)
                cmToAngle = -60 + (120 * ((pingValue * 0.01) - 0.25));
                // Scaling angles to tick values
                angleToTicks = 1500 + (cmToAngle * 15);
                // binds ping values to max and min pulse rates
                if (angleToTicks > RC_SERVO_MAX_PULSE)
                    angleToTicks = RC_SERVO_MAX_PULSE;
                if (angleToTicks < RC_SERVO_MIN_PULSE)
                    angleToTicks = RC_SERVO_MIN_PULSE;
                RCServo_SetPulse(angleToTicks);
              }
                break;
            case(ENCODER):
                rawAngle = RotaryEncoder_ReadRawAngle();
                // 0-360 deg conversion
                //Converts raw angle ticks to degrees. 360 / 16200 (largest deg/tick size) = 0.02222
                angleConversion = (rawAngle * .022);
                // converts from 0-360 deg => +-180 deg
                if (angleConversion > 179)
                    angleConversion = angleConversion - 360;
                // Linear interpolation is used to map the converted angles to values that fit the range of 600-2400us
                // normalization: (angle + (angle_difference / 2 )) / angle_difference. In this case angle_diff = 120 deg
                lerpRatio = (angleConversion + (ANGLE_DIFF / 2)) * ANGLE_DIFF_FRACTIONAL;
                // returns mapped values into motor us
                mappedAngle = lerp(RC_SERVO_MIN_PULSE,RC_SERVO_MAX_PULSE,lerpRatio);
                // Converts degrees to millidegrees and Big Endian
                angleReport.Angle = Protocol_IntEndednessConversion((signed int)(angleConversion * 1000));
                // Checks to see if the encoder angle is between the RC-SERVO range (+-60 deg)
                if (mappedAngle >= RC_SERVO_MIN_PULSE && mappedAngle <= RC_SERVO_MAX_PULSE)
                {
                    angleReport.status = IN_RANGE;
                    RCServo_SetPulse(mappedAngle);
                }
                else if (mappedAngle < RC_SERVO_MIN_PULSE)
                    angleReport.status = UNDER_RANGE;
                else if (mappedAngle > RC_SERVO_MAX_PULSE)
                    angleReport.status = OVER_RANGE;
                break;
        }
        // Send Angle Report at 20Hz
        milliseconds = FreeRunningTimer_GetMilliSeconds();
        if (milliseconds >= desired_ms)
        {
            desired_ms += 100;
            Protocol_SendMessage(5, ID_LAB2_ANGLE_REPORT, &angleReport);
        }
    }
    return 0;
}
```

5.0.2 Ping Sensor

```
/*
 * File:  PingSensor.c
 * Author: Mel Ho
 * Brief:  Functions for PingSensor
 * Created on February 1, 2021, 10:06 PM
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include "PingSensor.h" // The header file for this source file.
#include "BOARD.h"
#include <xc.h>
#include <sys/attribs.h>
#include "FreeRunningTimer.h"
#include "Protocol.h"
#include "string.h"
#include "stdio.h"
#include "MessageIDs.h"
#include "delays.h"
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define TIME_OF_FLIGHT 340
#define FPB (BOARD_GetPBClock())
#define TIMERFREQ_TR4 16.667
#define TIMERFREQ_TR2 50
#define PRESCALER_16 16
#define PRESCALER_64 64
#define CM_TO_MM 10
#define OUTPUT 0
#define INPUT 1
#define LOW 0
#define HIGH 1
#define FALSE 0
#define TRUE 1
#define TIMER_2 1
#define SIMPLE_CAPTURE_EVERY 0b110
#define FIRST_CAPTURE_RISE 1
#define INTERRUPT_EVERY_CAPTURE 0b00;
#define RISING_EDGE 1
#define FALLING_EDGE 0
#define TRIGGER_PIN_INIT (TRISDbits.TRISD8 = OUTPUT)
#define TRIGGER_PIN LATDbits.LATD8 //Pin 2
/*******************************************************************************
 * VARIABLES                                                                   *
 ******************************************************************************/
unsigned short uptime = 0;
unsigned short downtime = 0;
unsigned short timeInterval = 0;
unsigned short distance = 0;
unsigned int edgeState = RISING_EDGE;
static char message[MAXPAYLOADLENGTH];
static unsigned int milliseconds = 0;
static unsigned int Hz = 100;
//#define PING_TEST 1
/*******************************************************************************
 * PUBLIC FUNCTION IMPLEMENTATIONS                                             *
 ******************************************************************************/
/**
 * @Function PingSensor_Init(void)
 * @param None
 * @return SUCCESS or ERROR
 * @brief initializes hardware for PingSensor with the needed interrupts */
int PingSensor_Init(void)
{
    T4CON = 0;
    TMR4 = 0;
    T2CON = 0;
    TMR2 = 0;
    IC3CON = 0;
    IC3CONbits.ICTMR = TIMER_2; //TIMER2
    IC3CONbits.ICM = SIMPLE_CAPTURE_EVERY; //Edge Detect Mode - every edge
    IC3CONbits.FEDGE = FIRST_CAPTURE_RISE;
    IC3CONbits.ICI = INTERRUPT_EVERY_CAPTURE;
    IC3BUF; // clear input capture buffer
    IPC3bits.IC3IP = 2; //IC3 PRIORITY (higher than timer 2)
    IFS0bits.IC3IF = 0;
    IEC0bits.IC3IE = 1;
    T2CONbits.TCKPS = 0b100; // 1:16
    T4CONbits.TCKPS = 0b110; // 1:64
    // I set both PRs to my desired tick value, given the frequency I want that is then prescaled to fit a 16-bit timer.
    PR2 = FPB / TIMERFREQ_TR2 / PRESCALER_16; // interrupt every 20ms / 20000us
    PR4 = FPB / TIMERFREQ_TR4 / PRESCALER_64; // interrupt at 60ms (16.666667Hz)
    IPC4bits.T4IP = 1; // TIMER 4 PRIORITY
    IPC4bits.T4IS = 1;
    IPC2bits.T2IP = 3; // TIMER 2 PRIORITY
    IPC2bits.T2IS = 2;
    IFS0bits.T4IF = 0;
    IEC0bits.T4IE = 1;
    TRIGGER_PIN_INIT;
    TRIGGER_PIN = LOW;
    T4CONbits.ON = TRUE;
    T2CONbits.ON = TRUE;
    IC3CONbits.ON = TRUE;
    return SUCCESS;
}
/**
 * @Function int PingSensor_GetDistance(void)
 * @param None
 * @return Unsigned Short corresponding to distance in millimeters */
unsigned short PingSensor_GetDistance(void)
{
    // time = timeInteveral * (4 x Pre-scaler) / Fosc (number of ticks x time of each tick).
    // this converts from ticks to us.
    // distance  = (time/2) / 58 (convert to cm)
    unsigned short dist = ((timeInterval * ((4 * PRESCALER_16) / TIMERFREQ_TR2)) / 58 / 2);
    return (dist);
}
void __ISR(_TIMER_4_VECTOR) Timer4IntHandler(void)
{
    IFS0bits.T4IF = 0;
    TRIGGER_PIN = HIGH;
    delay_us(10);
    TRIGGER_PIN = LOW;
}
void __ISR(_INPUT_CAPTURE_3_VECTOR) __IC3Interrupt(void)
{
    IFS0bits.IC3IF = 0;
    switch(edgeState)
    {
        case RISING_EDGE:
            if (edgeState == PORTDbits.RD10)
            {
                uptime = (0xFFFF & IC3BUF);
                edgeState = FALLING_EDGE;
                // Clear buffer if it's still not empty
                IC3BUF;
            }
            break;
        case FALLING_EDGE:
            if (edgeState == PORTDbits.RD10)
            {
                downtime = (0xFFFF & IC3BUF);
                timeInterval = downtime - uptime;
                edgeState = RISING_EDGE;
                // Clear buffer if it's still not empty
                IC3BUF;
            }
            break;
    }
            IC3BUF;
}
#ifdef PING_TEST
int main(void)
{
    BOARD_Init();
    PingSensor_Init();
    FreeRunningTimer_Init();
    Protocol_Init();
    sprintf(message, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
    while(1)
    {
        milliseconds = FreeRunningTimer_GetMilliSeconds();
        if (milliseconds >= Hz)
        {
            Hz += 100;
            distance = PingSensor_GetDistance();
            distance = Protocol_ShortEndednessConversion(distance);
            Protocol_SendMessage(2, ID_PING_DISTANCE,&distance);
        }
    }
}
#endif
```

5.0.3 R/C Servo

```
/*
 * File:   RCServo.c
 * Author: Mel Ho
 * Brief:
 * Created on <month> <day>, <year>, <hour> <pm/am>
 * Modified on <month> <day>, <year>, <hour> <pm/am>
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include "RCServo.h" // The header file for this source file.
#include "BOARD.h"
#include <xc.h>
#include <sys/attribs.h>
#include "Protocol.h"
#include "FreeRunningTimer.h"
#include "string.h"
#include "stdio.h"
#include "MessageIDs.h"
#include "delays.h"
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define FPB (BOARD_GetPBClock())
#define TIMEFREQ_TR3 20 //20Hz
#define TIMER_PRESCALE_32 32 // 1:32
#define FALSE 0
#define TRUE 1
#define TIMER_3 1
#define TIMER_3_PULSE FPB / TIMEFREQ_TR3 / TIMER_PRESCALE_32 // 62500 ticks
#define MS_PERIOD 50 // 50ms
#define CENTER_PULSE (TIMER_3_PULSE / MS_PERIOD) * (RC_SERVO_CENTER_PULSE * .001)
#define MIN_PULSE (TIMER_3_PULSE / MS_PERIOD) * (RC_SERVO_MIN_PULSE * .001)
#define MAX_PULSE (TIMER_3_PULSE / MS_PERIOD) * (RC_SERVO_MAX_PULSE * .001)
//#define SERVO_TEST 1
static unsigned int prevPulse;
static unsigned int inPulseStatus = RC_SERVO_CENTER_PULSE;
unsigned int status;
static char message[MAXPAYLOADLENGTH];
/*******************************************************************************
 * PUBLIC FUNCTION IMPLEMENTATIONS                                             *
 ******************************************************************************/
/**
 * @Function RCServo_Init(void)
 * @param None
 * @return SUCCESS or ERROR
 * @brief initializes hardware required and set it to the CENTER PULSE */
int RCServo_Init(void)
{
    T3CON = 0;
    TMR3 = 0;
    T3CONbits.TCKPS = 0b101; // 1:32 prescale
    PR3 = TIMER_3_PULSE;
    IPC3bits.T3IP = 3; // Timer 3 priority
    IPC3bits.T3IS = 2;
    OC3CON = 0;
    OC3CONbits.OCTSEL = TIMER_3;
    OC3CONbits.OCM = 0b101;
    OC3RS = CENTER_PULSE; //1500us default orientation is center
    IFS0bits.OC3IF = 0; //OC3 priority
    IEC0bits.OC3IE = 1;
    T3CONbits.TON = TRUE;
    OC3CONbits.ON = TRUE;
    return SUCCESS;
}
/**
 * @Function int RCServo_SetPulse(unsigned int inPulse)
 * @param inPulse, integer representing number of microseconds
 * @return SUCCESS or ERROR
 * @brief takes in microsecond count, converts to ticks and updates the internal variables
 * @warning This will update the timing for the next pulse, not the current one */
int RCServo_SetPulse(unsigned int inPulse)
{
    if (inPulse > RC_SERVO_MAX_PULSE || inPulse < RC_SERVO_MIN_PULSE)
        return ERROR;
    if (inPulse != prevPulse)
    {
        prevPulse = inPulse;
        // conversion from timer ticks to microseconds
        OC3RS = (TIMER_3_PULSE / MS_PERIOD) * (inPulse * 0.001);
    }
    return SUCCESS;
}
/**
 * @Function int RCServo_GetPulse(void)
 * @param None
 * @return Pulse in microseconds currently set */
unsigned int RCServo_GetPulse(void)
{
    return prevPulse;
}
/**
 * @Function int RCServo_GetRawTicks(void)
 * @param None
 * @return raw timer ticks required to generate current pulse. */
unsigned int RCServo_GetRawTicks(void)
{
    return (TIMER_3_PULSE / (MS_PERIOD * (prevPulse * .001)));
}
void __ISR(_OUTPUT_COMPARE_3_VECTOR) __OC3Interrupt(void)
{
    IFS0bits.OC3IF = 0;
}
/*******************************************************************************
 * PRIVATE FUNCTION IMPLEMENTATIONS                                            *
 ******************************************************************************/
#ifdef SERVO_TEST
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    RCServo_Init();
    sprintf(message, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
    unsigned int inPulse;
    while(1)
    {
        if (Protocol_ReadNextID() == ID_COMMAND_SERVO_PULSE)
        {
            Protocol_GetPayload(&inPulse);
            inPulse = Protocol_IntEndednessConversion(inPulse);
            RCServo_SetPulse(inPulse);
            inPulseStatus = RCServo_GetPulse();
            inPulseStatus = Protocol_IntEndednessConversion(inPulseStatus);
            Protocol_SendMessage(4, ID_SERVO_RESPONSE, &inPulseStatus);
        }
    }
}
#endif
```

5.0.4 Free Running Timer

```
/*
 * File:   FreeRunningTimer.c
 * Author: Mel Ho
 *
 * Created on December 19, 2012, 2:08 PM
 */
#ifdef _SUPPRESS_PLIB_WARNING
#undef_SUPPRESS_PLIB_WARNING
#endif
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include <BOARD.h>
#include <xc.h>
```

```
#include "FreeRunningTimer.h"
#include <sys/attribs.h>
#include "Protocol.h"
#include "delays.h"
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define FPB (BOARD_GetPBClock())
#define TIMERFREQ 1000
//#define TIMER_TEST 1
static unsigned int milliseconds = 0;
static unsigned int microseconds = 0;
/*******************************************************************************
 * PUBLIC FUNCTION IMPLEMENTATIONS                                             *
 ******************************************************************************/
/**
 * @Function TIMERS_Init(void)
 * @param none
 * @return None.
 * @brief  Initializes the timer module */
void FreeRunningTimer_Init(void){
    T5CON = 0;
    TMR5 = 0;
    PR5 = FPB / TIMERFREQ / 4;
    T5CONbits.TCKPS = 0b010;
    IPC5bits.T5IP = 3;
    IPC5bits.T5IS = 1;
    IFS0bits.T5IF = 0;
    IEC0bits.T5IE = 1;
    T5CONbits.ON = 1;
}
/**
 * Function: TIMERS_GetMilliSeconds
 * @param None
 * @return the current MilliSecond Count
   */
unsigned int FreeRunningTimer_GetMilliSeconds(void)
{
    return milliseconds;
}
/**
 * Function: TIMERS_GetMicroSeconds
 * @param None
 * @return the current MicroSecond Count
   */
unsigned int FreeRunningTimer_GetMicroSeconds(void)
{
    return (microseconds);
}
void __ISR(_TIMER_5_VECTOR, ipl3auto) Timer5IntHandler(void)
{
    IFS0bits.T5IF = 0;
    milliseconds++;
    microseconds += 1000;
}
#ifdef TIMER_TEST
#include "FreeRunningTimer.h"
#include <stdio.h>
#include "Protocol.h"
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    FreeRunningTimer_Init();
    char testMessage[MAXPAYLOADLENGTH];
    unsigned int currentMilli = 0;
    unsigned int currentMicro = 0;
    TRISDbits.TRISD8 = 0;
    LATDbits.LATD8 = 0;
    sprintf(testMessage, "Free Running Timer Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(testMessage);
    while(1)
    {
        if (FreeRunningTimer_GetMilliSeconds() % 2000 == 0) {
            LEDS_SET(0xFFFF);
            LATDbits.LATD8 ^= 1;
            currentMicro = FreeRunningTimer_GetMicroSeconds();
            currentMilli = FreeRunningTimer_GetMilliSeconds();
            sprintf(testMessage,"ms: %d\tus: %d\r\n",currentMilli, currentMicro);
            Protocol_SendDebugMessage(testMessage);
            delay_ms(1);
        }
        else
```

```
            LEDS_SET(0x0000);
    }
}
#endif
```

5.1 B. Video Footage

https://drive.google.com/drive/folders/18K2VXPV8ZR2Et93Jx7W68WGZ50cDFJsx?usp=sharing