Johnson Le

February 7, 2019

CMPE 167/L

Lab 2 Report

Introduction

A big part of this lab is utilizing interrupts to process changes in events. The events handled in this lab are state changes in pins and timers. A lot of time will be spent reading datasheets as well as provided files.

Part 1 – Read QEI and Generate Angle Accumulation

The quadrature encoder interface (QEI) is split into two main sides, the encoder side and the LED controller side. Part 1 focuses on the encoder and triggering interrupts when the state of the encoder is changed.

Method:

The first step is to wire up the encoder. Connecting pin A and B on the encoder to pin 36 and 37 on the uno32, the provided files will be able to interrupt if any of the two pins change state. This is a digital signal so there are only two states per pin totaling 4 states total between the two. It is important to see how the states flow from one to the other to determine the direction.

Displaying the states of the pins 36 and 37 individually on the OLED allows the user to visually see the state the pins and how they change as the encoder rotates. This may be different for each encoder. Once the flow of states is determined, a state machine can be built and a count can be determined. Turning clockwise increments the count and counterclockwise decrements.

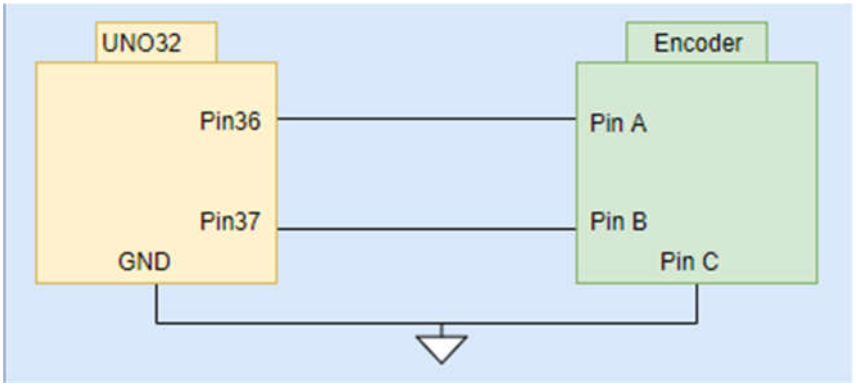

Fig 1 Wiring for Encoder

Results:

The encoder I was using idles with both states high. The transitions of the encoder going clockwise is 11->10->00->01->11. Using this this information, a basic state machine (Fig 2) was built incrementing a counter for each transition clockwise and decrementing for every transition

counterclockwise. Marking the encoder with a sharpie, I was able to rotate the encoder a full 360 and noticed a count of 96 confirming what was mentioned in the lab manual.

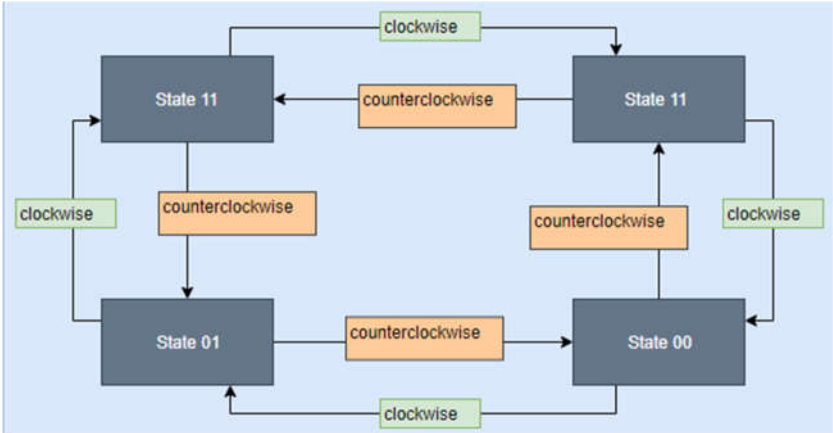

Fig 2 Encoder State Machine

Part 2 – Map QEI to Color

Now that we have decoded the encoder and have a function that returns the current count, we are able to utilize that to program the LED colors based on the count.

Method:

The first step is to solder the encoder onto the provided PCB. There are three pins on the QEI to control the color and a fourth pin to provide power or a connection to ground. There are two options to wiring this up. This can be done with a pull-up or pull-down resistor. We will be using the pull-up method connecting the pin denoted "+" on one side but "gnd" on the other. The three pins denoted "R", "G", and "B" will have a resistor in-between itself and a pwm pin from the uno32. To access the pins, multiple datasheets must be read to figure out which pins provide pwm signals through the provided pwm files. There are 5 pwm pins listed.

There are multiple color wheels available. For this lab, the one provided on the lab manual was used. From part 1, a count of 96 indicates that the encoder was rotated 360 degrees clockwise. Counterclockwise, the minimum is -96. Mapping the range of -96 to 96 onto -360 to 360 makes it easier to denote which angle will correspond to what color.

https://color.adobe.com/create/color-wheel/

There are multiple websites out there, such as the one provided above, that will help with determining color based on the intensity of red, green, or blue.

Results:

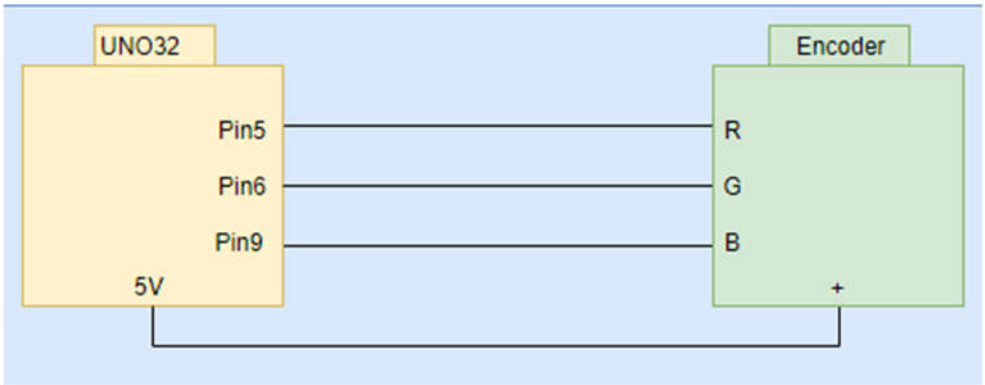

Fig 3Led Pin Diagram

After diving around datasheets, I used pins 5,6, and 9 for pins R, G, and B respectfully. Once connected, I did some testing to see if I could force a certain color on the LED. Utilizing the link mentioned earlier, I was able to determine intensity levels for the colors on the color wheel. Intensity level is determined by the duty cycle. Because I used the pull-up configuration, the pins start high with a duty cycle of 0. Therefore, I had to invert the duty cycle values. Using excel for organizing, I noted down what duty cycle I needed for each pin depending on the angle.

COLOR	ANGLE	RED	GREEN	BLUE
orange	-360	0	500	1000
red	-315	0	1000	1000
purple	-270	500	1000	0
blue	-225	1000	1000	0
light blue	-180	1000	500	0
teal	-135	1000	0	0
lime	-90	1000	0	500
green	-45	1000	0	1000
yellow	0	0	0	1000
orange	45	0	500	1000
red	90	0	1000	1000
purple	135	500	1000	0
blue	180	1000	1000	0
light blue	225	1000	500	0
teal	270	1000	0	0
lime	315	1000	0	500
green	360	1000	0	1000

Fig 4Angle to Color Chart

I just did a bunch of if else statements to control the color. For the fading affect, I made each increment of count change slightly into the next. For example, going counterclockwise from 0 to -45 degrees, red changes from 0 to 1000 but green and blue stay the same. Therefore, I made red

increase slowly. This is done by mapping the angles where red is changing which is 0 to -45 onto 0 to 1000. Although this method is tedious, it is simpler to think about.

Part 3 – Ping the Ping Sensor and Capture Output

Moving away from the QEI, part 3 explores the use of a common ultrasonic sensor. This is used to detect an object at a distance by sending out small signals that will bounce back off a surface and the sensor waits to receive that signal.

Method:

Reading through the provided PING.h file, it is noted that the echo pin is set-up to be on pin 34. A change on pin 34 will trigger an interrupt. A trigger pin has not been designated and will need to be. To set a pin for input or output, use TRIS register. To write a pin high or low, use LAT register. To read the status of a pin, use PORT register. For example, pin 35 is on register D11 so TRISDbits.TRISD11=0, LATDbits.LATD11=1, and PORTDbits.RD11 will set the pin as an output, set it high, and read the state of the pin respectively.

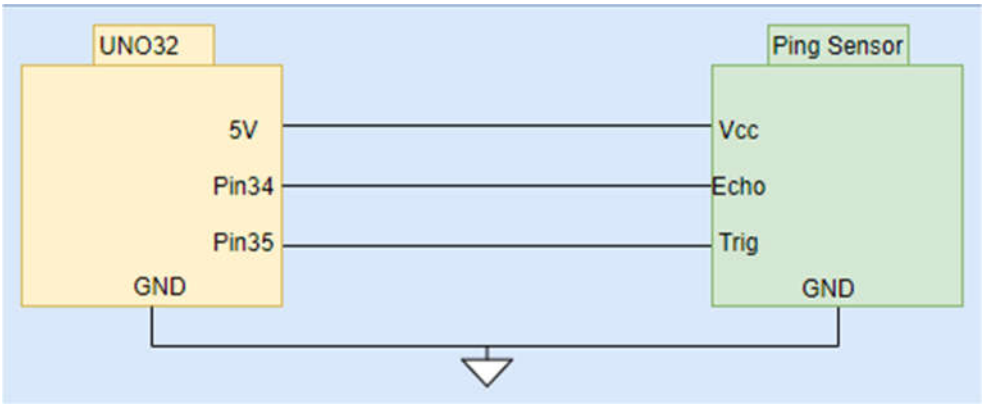

Fig 5Ping Sensor Pin Diagram

PING.h sets us up with some initial code. This comes with two interrupts, a hardware trigger and timer interrupt. We want to set the ping's trigger pin high for at least 10us and wait for 60ms then repeat so this is done through timer interrupts. The lab manual mentions that the uno32 runs a peripheral bus clock of 40MHz with a certain pre-scale. The pre-scale value used is 1:64. With some calculations, changing the period to 6.25 will provide 10us and period to 37000 will provide 60ms.

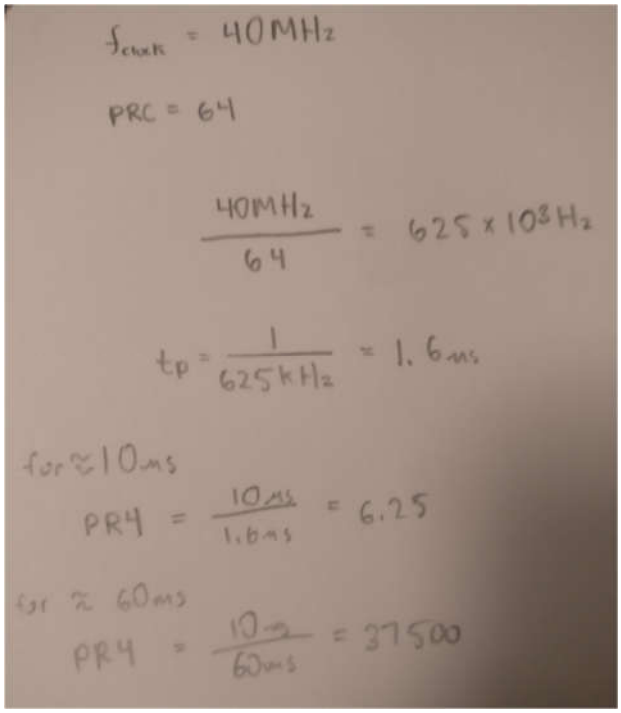

Fig 6Timer Interrupt Calculations

The period for the timer can be set by PR# where # = the timer used. Using this information, we can control when the next timer interrupt will occur allowing for precise trigger control. This is done by a simple state machine.

Once trigger is set up, the echo can be received and processed through the Change Notice interrupt. The time echo stays high depends on the time it takes to read the signal. We can use this to calculate distance based on the time echo pin stays high.

Results:

As mentioned in the method, I used pin 35 for my trigger pin. Once the timer interrupt was all set up for controlling the trigger pin, I displayed the output onto an oscilloscope to confirm.

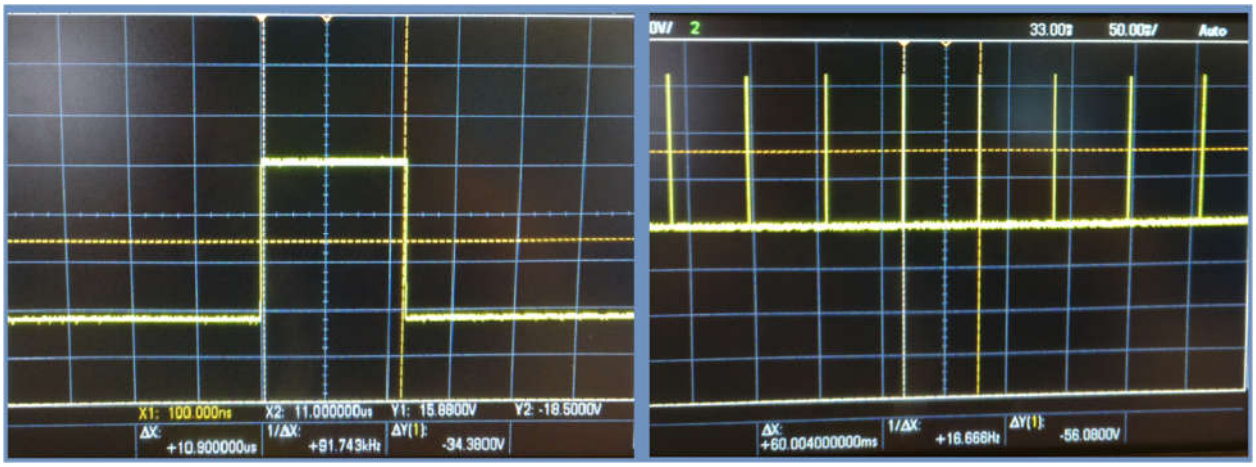

Fig 7Trigger Pin Trace

Connecting this to the trigger pin, I wanted to check to see if echo was receiving. This is done by now connecting the echo pin to the oscilloscope. It can be observed that the length that echo stays on is proportional to the distance it takes to receive a signal. Using a state machine inside the interrupt to handle the echo's pin state, I calculated the time the echo stays high providing me with the time of flight. From the datasheet, dividing this value by 5.8 provides centimeters. I divided the time of flight by 58 to acquirer millimeters. I displayed this value on the OLED to prepare for part 4.

Part 4 – Calibrate the Ping Sensor Using Least Squares

Method:

Utilizing a tape measurer, mark out some points. Note down expected distance seen by the tape measure with the experimental distance seen on the uno32. Once sufficient data has been taken, use Matlab to calculate least squares.

Result:

Expected(mm)	Experimetal(mm)
50	45
100	100
150	149
200	202
250	254
300	304
350	355
400	403
450	455
500	503
550	554
600	605
650	655
700	703
750	753
800	803
850	852
900	901
950	955
1000	1001

Fig 8Ping Sensor Data

The expected vs experimental data I had was surprisingly similar. Therefore, using the raw data would've been fine.

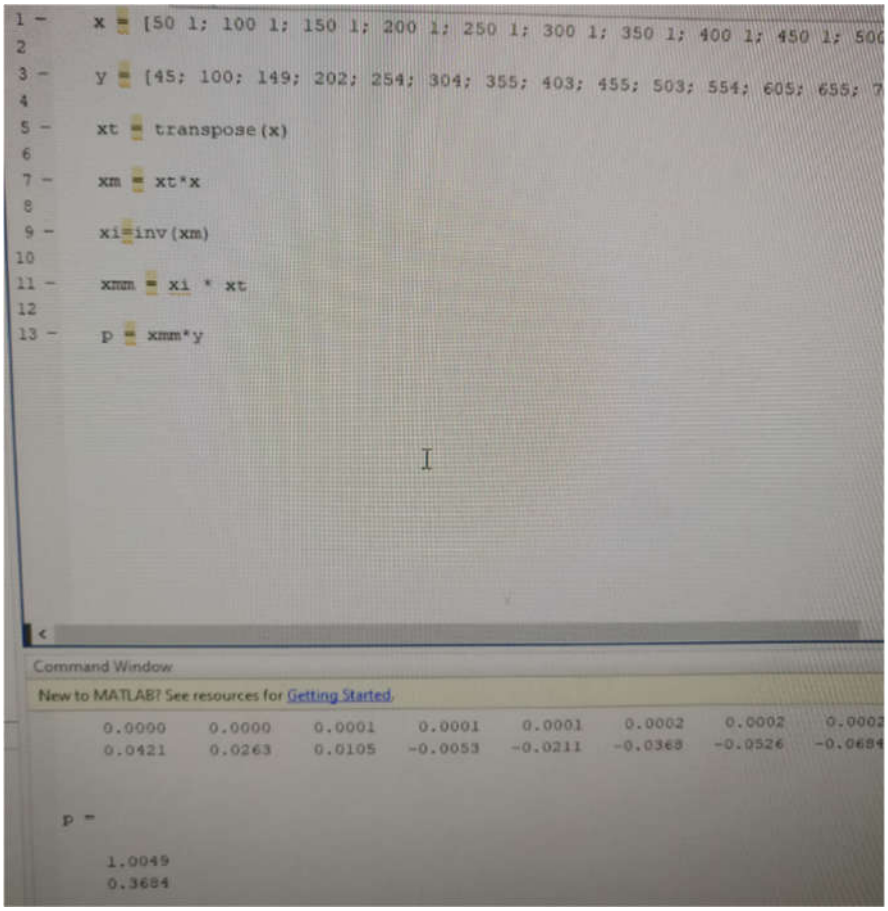

Fig 9Matlab Code with Output

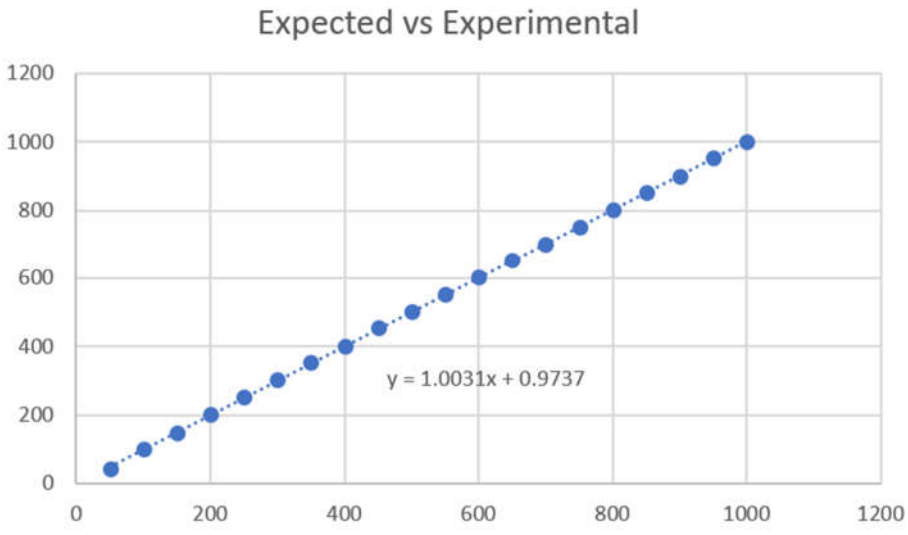

Fig 10Ping Sensor Data Graph

Utilizing Matlab, a slope and intercept is calculated, y = 1.0031x + 0.9737. Here, x is the expected and y is the experimental. Solving for x, we can plug in the experimental value for y to get the desired distance.

Part 5 – Tone Out Based on Distance

Method:

Now that we have a distance measurement in code, we can map this value to a tone for the speaker. The setup is the same as in previous labs, so a little review helped. Make sure to set the tone to a nice range. Map a range of distance to a range of tone and output to the audio amplifier.

Result:

I restricted the distance to a maximum of 1000mm. I then mapped the distance value of 0 to 1000 onto 100 to 800 for the tone. Using an average between the last 5 distance values and hysteresis for filtering, I managed to get very smooth sound transitions depending on the distance.

Part 6 – RC Time Constant Based Measurement

Method:

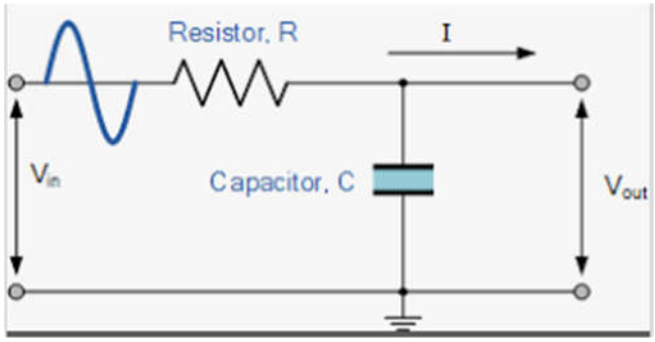

Fig 11RC Circuit

Taking a look at Figure 11, the resistor used is 100k and the capacitor is the touch sensor. The incoming signal will be created by a signal generator. Using square waves and altering the frequency of the signal, we can observe the behavior of the touch sensor. Use an oscilloscope to analyze the trace. Note down peak to peak values when touching and not touching the sensor while altering the frequency of the input. Using the peak to peak values, the time constant can be located by finding how long it takes the output to ramp up to (pk-to-pk * 0.63).

Result:

I noticed that the touch sensor only notices touch on one side depending on the polarity. Once wired up, I hooked both the input and output of the circuit for observations.

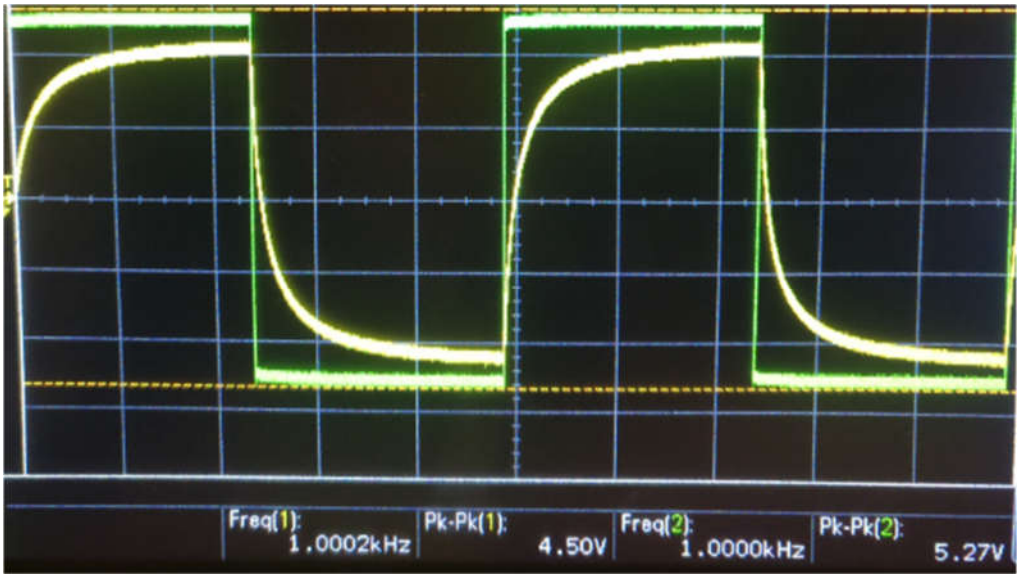

Fig 12 Trace of RC Circuit

The green trace is the input and the yellow trace is the output. This output trace shown is when the sensor is pressed. I took some data on excel seen on Figure 13 and 14.

Freq(Hz)	Pk-to-Pk	Pk-to-Pk * 0.63	time const(s)	Capacitance(F)
5000	5.11	3.2193	2.80E-06	2.80E-11
10000	5.12	3.2256	3.42E-06	3.42E-11
15000	5.11	3.2193	2.80E-06	2.80E-11
20000	5.11	3.2193	2.80E-06	2.80E-11

Fig 13 Non-Touching Data

Freq(Hz)	Pk-to-PK	Pk-to-Pk * 0.63	time const(s)	Capacitance(F)
5000	4.79	3.0177	1.68E-05	1.68E-10
10000	4.34	2.7342	1.58E-05	1.58E-10
15000	3.8	2.394	1.28E-05	1.28E-10
20000	3.34	2.1042	1.13E-05	1.13E-10

Fig 14 Touching Data

Analyzing figure 13, we can see the capacitor value calculated stays relatively the same which makes sense. When touching the sensor, the sensor should change in capacitance which is seen in figure 14.

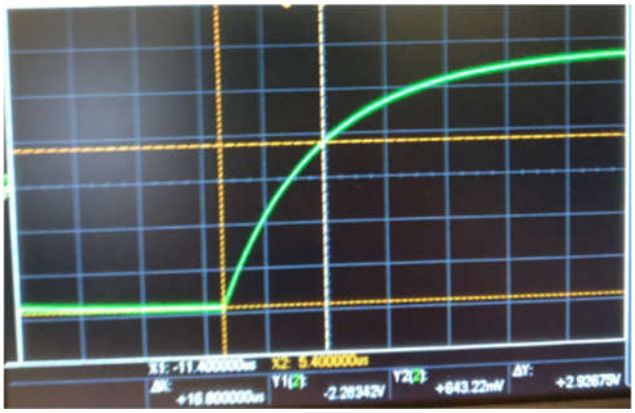

Fig 15 Finding Time Constant

Figure 15 is an example of how I found the time constant.

Part 7 – Capacitive Bridge and Difference Amp

Method:

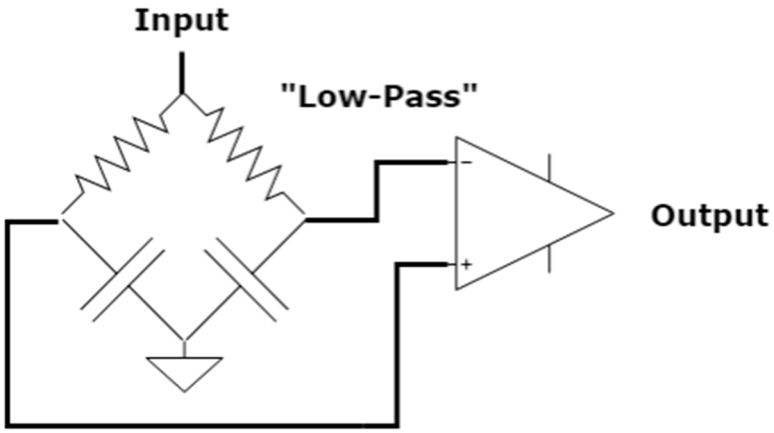

Fig 16 Low Pass Wheatstone Bridge

Starting with the low-pass Wheatstone bridge circuit, hookup the signal generator to the input in the circuit shown in figure 15. The two resistors used were both 100k ohm resistors with the capacitors both at 100pF. The touch sensor will be wired parallel to one of the capacitors. The voltage at the two terminals will differ and the difference can be seen at the output. Explore the traces to see how the two inputs to the op amp differ on the oscilloscope.

For the high-pass, do the same but the resistors and capacitors are flipped. Decide which circuit to use before moving on. Modify the gains for the two inputs into the difference amplifier to get a nice change in voltage when touching vs not touching.

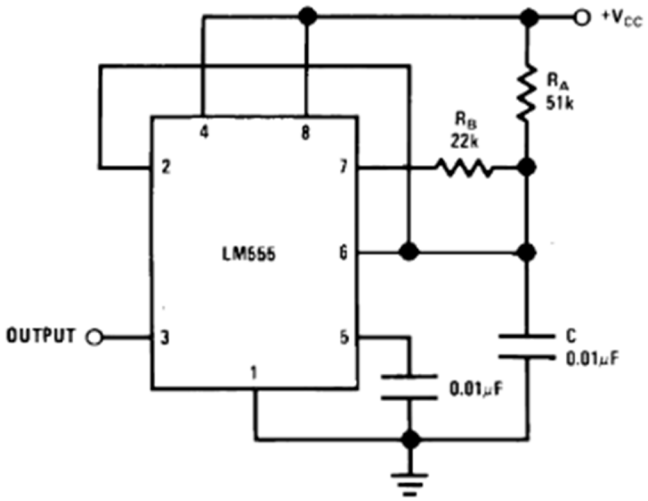

Fig 17 50% Duty Cycle Oscillator

Moving onto the LM555, set up a 50% duty oscillator. Set this as the input to the Wheatstone bridge circuit.

Result:

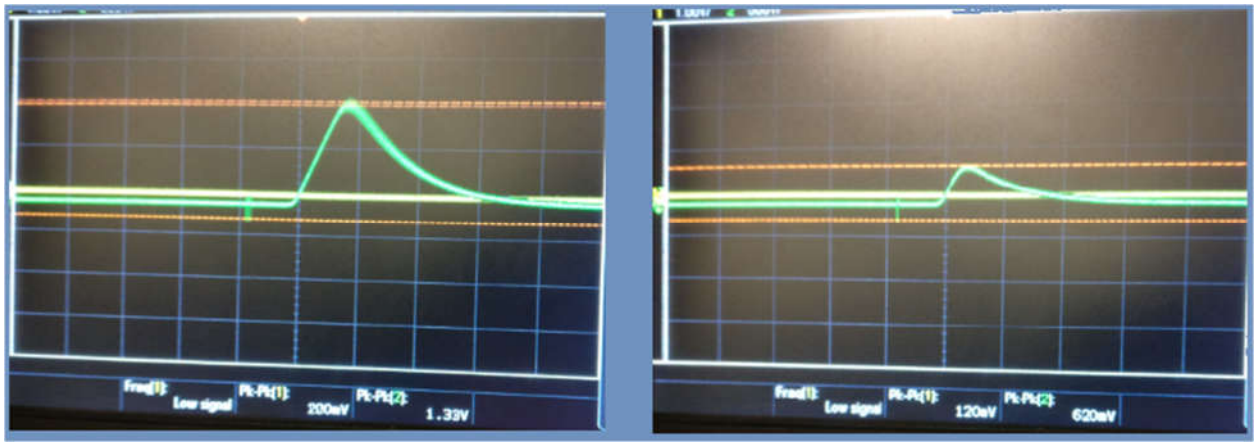

Fig 18 Low-Pass Wheatstone Trace

Pressing the sensor changes the amplitude of the signal. I ended up going with the high pass. I liked that it decayed from the peak slower. I also got nicer change in amplitude as seen in the trace below.

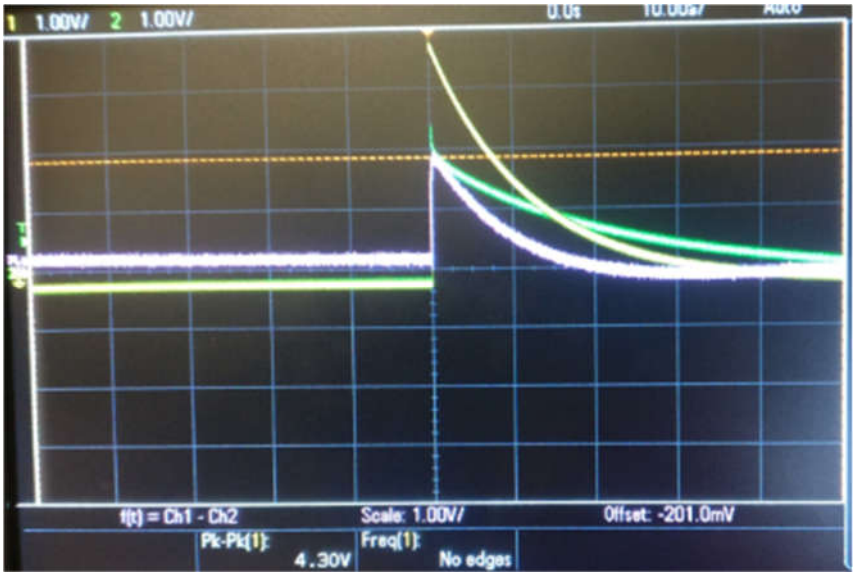

Fig 19 High Pass Wheatstone Bridge Trace

The red trace indicates the difference between the two sides of the Wheatstone bridge. This was a significant jump, so I kept with it. I was able to take the output of the differential amplifier and display the A/D value out to the Oled. I then coded up a hysteresis bound to indicate if the touch sensor was detected.

Part 8 – Relaxation Oscillator

Method:

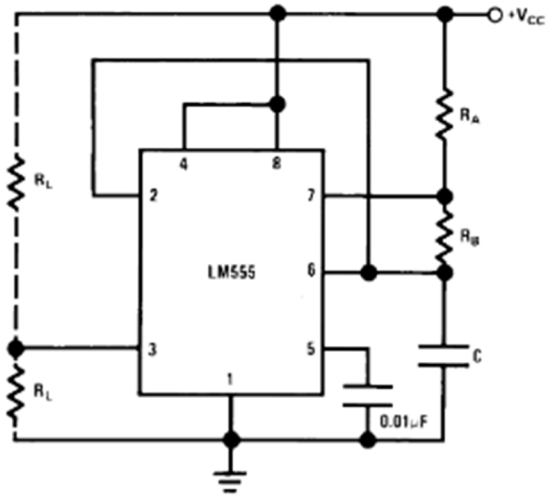

Fig 20 LM555 Astable Mode

Wire up figure 20. The load resistors on the left can be omitted. The C towards the bottom right is where the touch sensor is placed in parallel with a 22pF. Ra in my configuration is 3.9k ohms and Rb = 1.2M ohms giving near 50% duty cycle oscillating at about 11khz. The output should change in frequency as the capacitance is changed.

Result:

Displaying the trace of the LM555, I observe the output to change in period when I took the sensor, but the duty cycle does stay the same. In software, I used the input capture interrupt to grab the time when the interrupt occurs. I used IC4BUF to calculate period. I took an average of 50 values for more accuracy along with hysteresis bounds to make sure the uno32 knows when the sensor is touched.

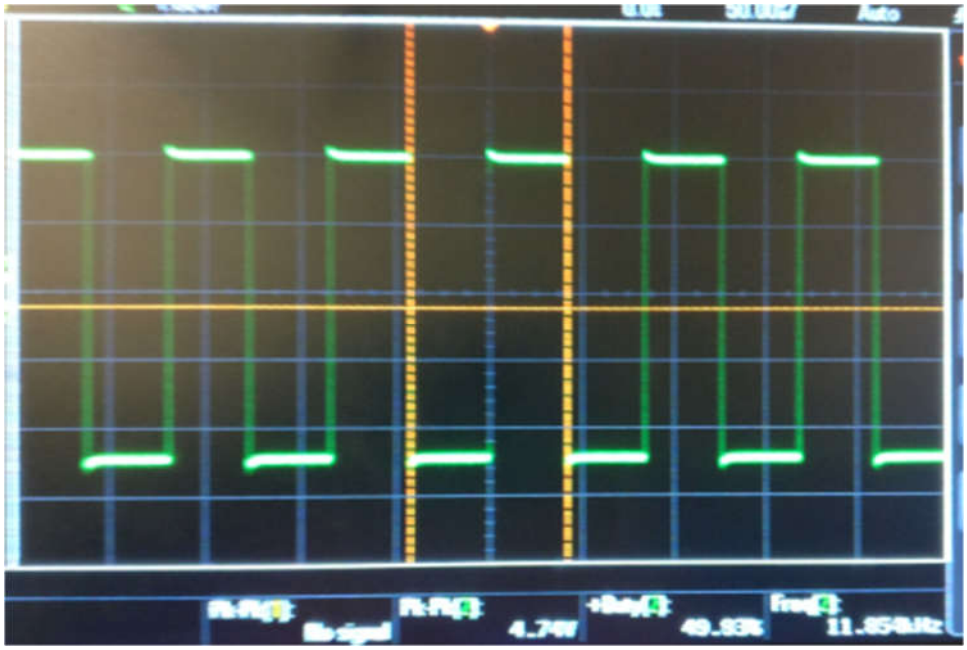

Fig 21 LM555 Output Sensor Not Touched

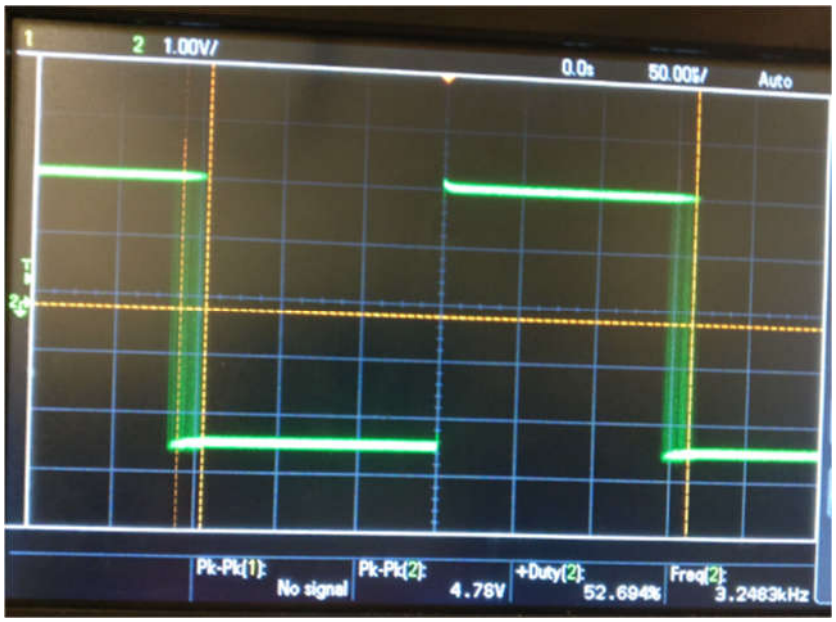

Fig 22 LM555 Output When Sensor is Touched

Conclusion:

This lab interfaces with many new sensors and modules which include the QEI, ping sensor, touch sensor, and LM555. I found this lab to be very interesting especially with the use of interrupts. I've always been interested in threading and non-blocking code. I felt that finding the pin registers was very annoying and think we should be given some IO library to interface the pins easier. This lab was much more time consuming and difficult as compared to the previous especially with all the new concepts of interrupts. All in all, I managed my time starting earlier and believe I have a deeper understanding of how interrupts work.