Leonid Shuster

ECE 167

2/12/20

Lab Report 2

**Introduction**

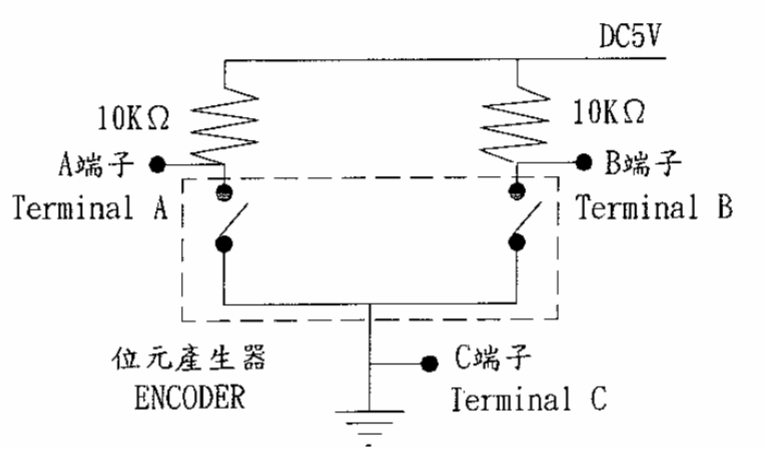
In this lab, we were tasked with learning how to use a Uno32 with a quadrature encoder, time based sensor, and capacitive sensor. In the first part, we set up a quadrature encoder to output position and direction of rotation, and mapped its rotation to a color wheel that would be displayed on the encoder’s RGB LED. In the second part, we used a ping sensor to determine distance from the sensor, which would then be used to play a tone on a speaker based on the distance. And in the third part, we set up a capacitive touch sensor to detect touch.

**Part 1: QEI**

In the first part of the lab, we were told to use a quadrature encoder to measure angular displacement, and map the position and direction of rotation to a color wheel that would then display its color on the encoder’s RGB LED. The first part consisted of the following steps:

**1.1**

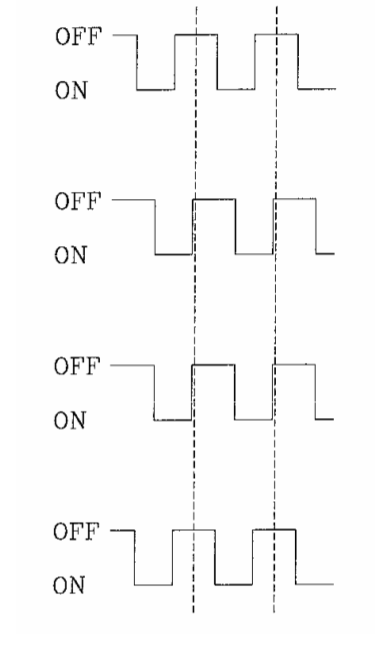
The first thing we had to do was set up the encoder onto our breadboard. After soldering the encoder onto its PCB, I looked in the encoder’s datasheet and found the following test circuit diagram:



After setting up the circuit, I hooked up the encoder’s outputs, terminals A and B, to an oscilloscope, and saw that when I rotated the knob, I saw two signals, one ahead of the other by 90 degrees of phase, that matched the expected outputs shown on the datasheet.

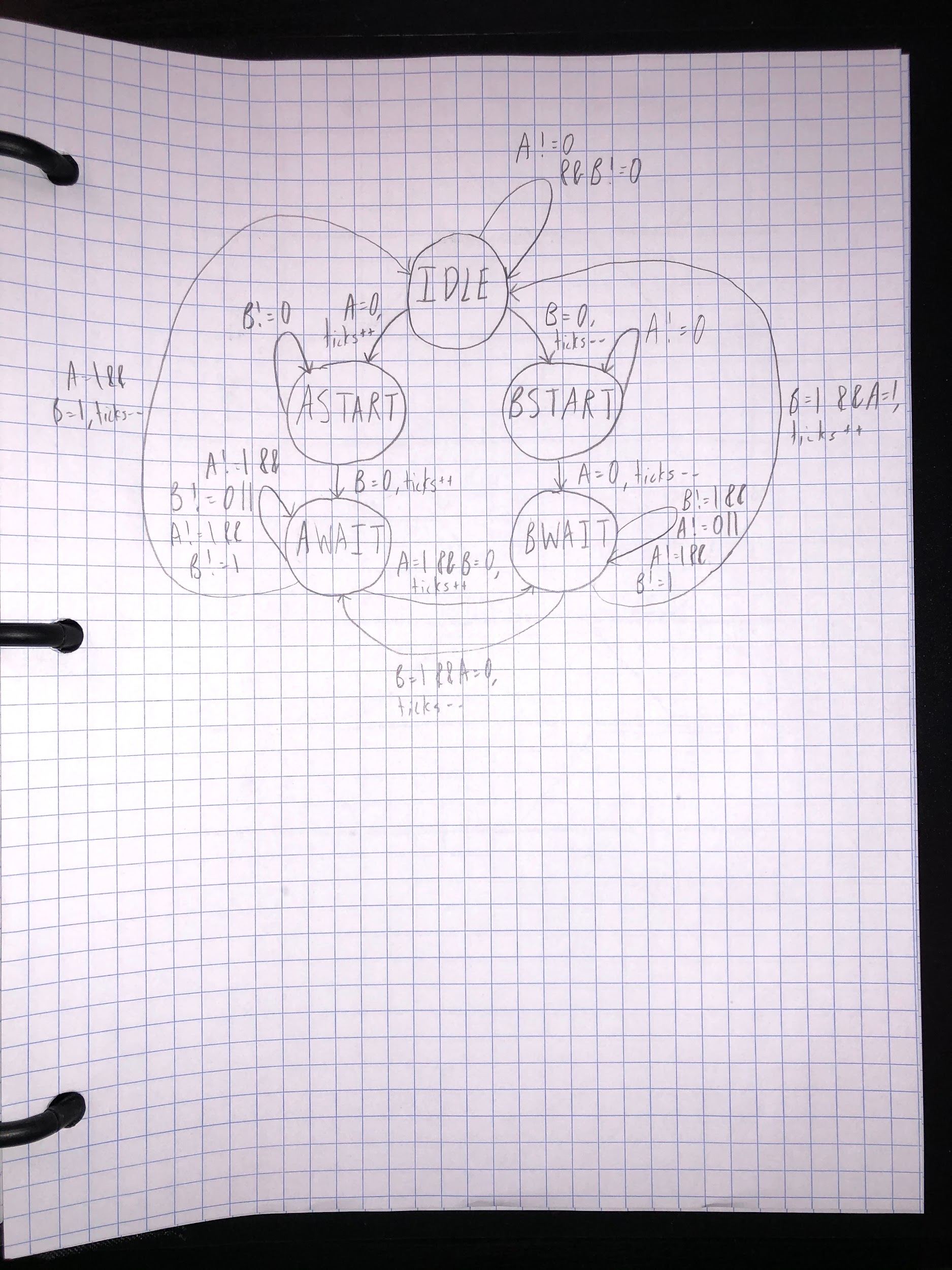
**1.2**

The next thing we had to do was create software that determined the position and direction of rotation. I looked at the datasheet of the encoder, and referred to the pictures that showed a normal cycle of a clockwise or counter-clockwise turn (see Figure 1).



*Figure 1*

By using interrupts that detected changes in the encoder’s outputs, terminals A and B, and a state machine within the interrupt that determined what part of the signal the interrupt was triggered in, I was able to figure out the position and direction of rotation (see Figure 2).



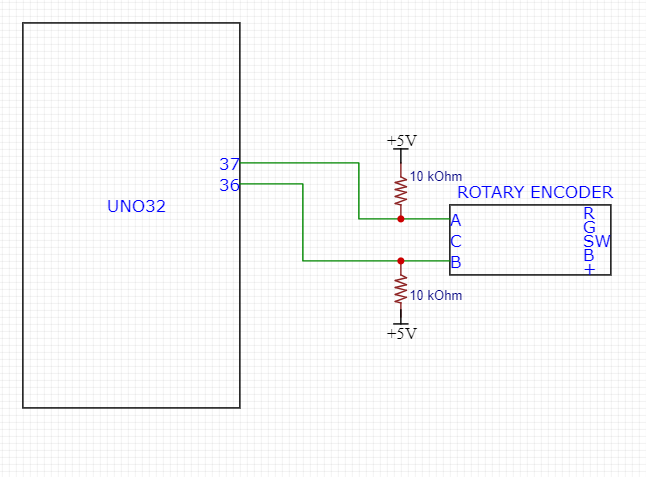
*Figure 2*

Essentially, the state machine began with both A and B being high. Depending on the direction of rotation, one of two things could happen:

1. If rotating clockwise, A would become low, the state machine would increment a variable that stored the number of ticks of rotation, and enter the state ASTART. The state machine would stay in ASTART until B also became low, and then increment the tick variable and move to state AWAIT. In AWAIT, the state machine waited for A to go high while B was still low, increment the variable, and then move to state BWAIT. In this final state, the state machine waited for B to go high while A was still high, increment the variable one more time, and then go back to IDLE, concluding one cycle of a clockwise turn.
2. If rotating counter-clockwise, B would become low, the state machine would decrement the variable that stored the number of ticks of rotation, and enter the state BSTART. The state machine would stay in BSTART until A also became low, and then decrement the tick variable and move to state BWAIT. In BWAIT, the state machine waited for B to go high while A was still low, decrement the variable, and then move to state SWAIT. In this final state, the state machine waited for A to go high while B was still high, decrement the variable one more time, and then go back to IDLE, concluding one cycle of a counter-clockwise turn.

The maximum number of ticks in a revolution was 96 or -96 (depending on clockwise or counter-clockwise rotation), so when the count exceeded those numbers, I reset the number of ticks back to 0. Then, to find the position in degrees of a rotation, I multiplied the tick count by the number of degrees in a single revolution, 360, and divided by the number of ticks in a single revolution, 96.

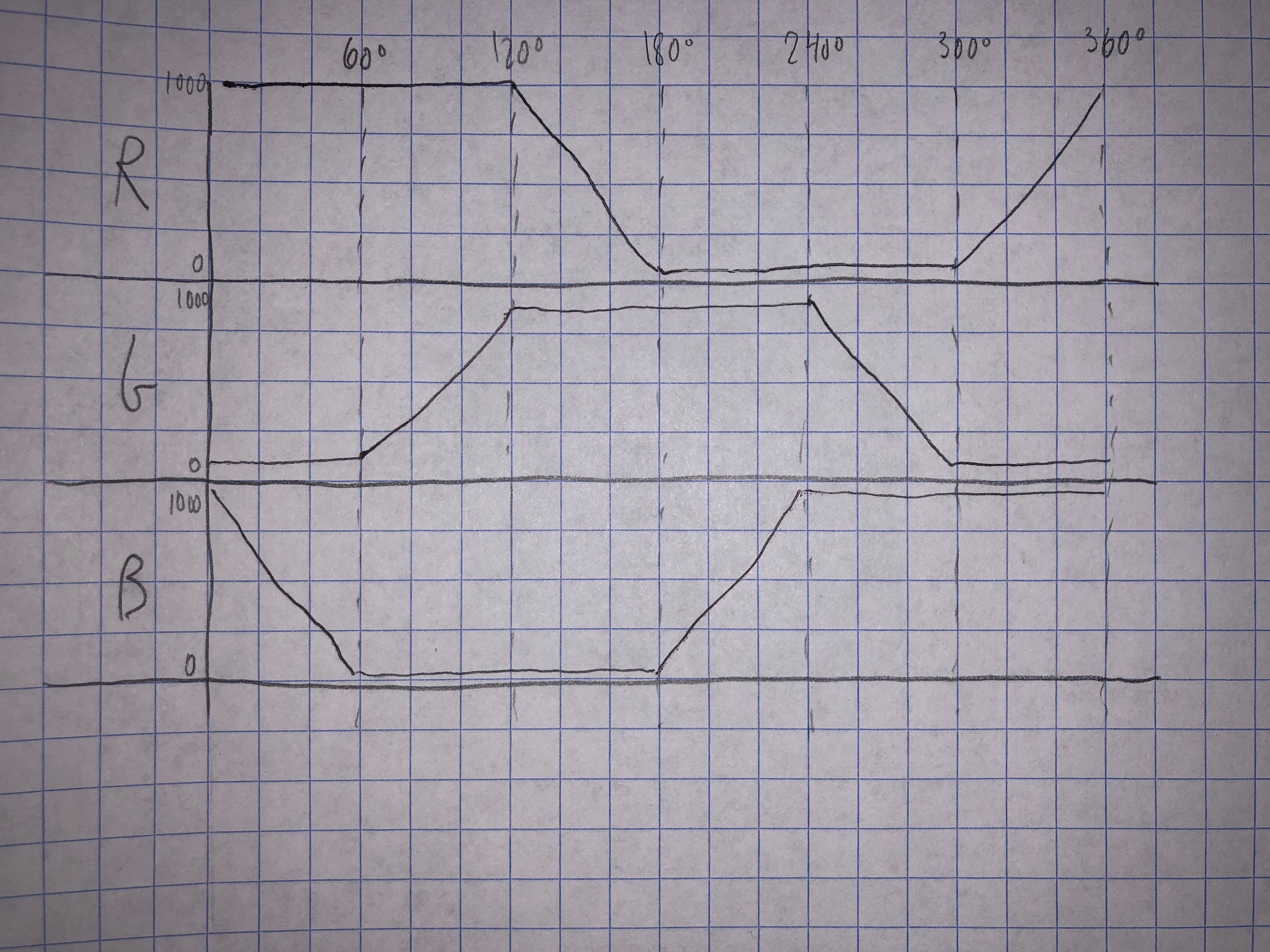
After implementing all of this in software, I connected the encoder to the Uno32 with terminals A and B of the encoder going to the two interrupts pins on the Uno32 (see Figure 3).



*Figure 3*

**1.3**

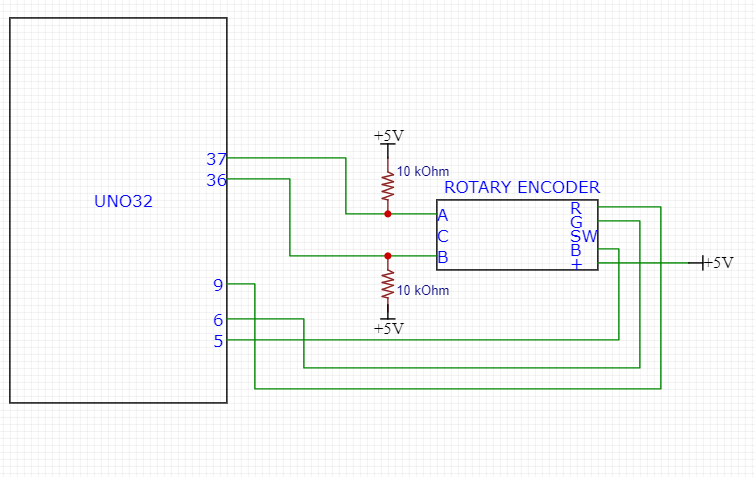
The last thing we had to do with the encoder was map its rotation to a color wheel that would be displayed on the encoder’s RGB LED. To do so, I looked at the following diagram that mapped the degrees of rotation to the duty cycle that would be sent out of each PWM pin to the encoder’s RGB inputs (see Figure 4).



*Figure 4*

I calculated equations of the duty cycle of each color in each range of degrees, and used them to vary the duty cycle based on the degrees of rotation. This diagram was only for clockwise rotation though, so to find the duty cycles going counter-clockwise, I just followed the reverse of the diagram for ranges of negatives degrees.

On the outside, I had the same configuration as before where terminals A and B of the encoder were connected to interrupt pins on the Uno32 and added three wires that connected the PWM pins on the Uno32 to the encoder’s RGB inputs (see Figure 5).



*Figure 5*

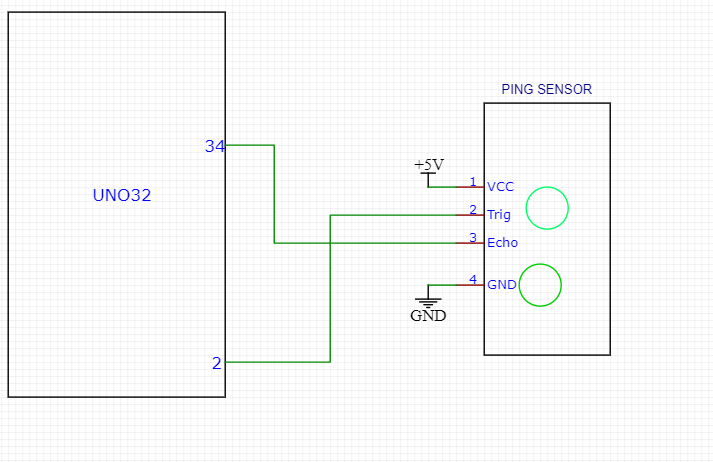
**Part 2: Ping Sensor**

In the next part of the lab, we were told to use a ping sensor to measure distance from the sensor, and then play a tone on a speaker based on the distance. The second part consisted of the following steps:

**2.1**

The first thing we had to do was create software that would trigger the sensor every 60 milliseconds. I configured a timer to generate interrupts every 60 milliseconds by changing the value of its period match register, and in that interrupt, I made the signal going to the sensor’s trigger pin high which would cause the sensor to then send out a ping. I also set up another interrupt that triggered whenever the output from the sensor’s echo pin changed. When the sensor sent out a ping, the echo output went high and the interrupt would be triggered, and when the sensor received its ping again, the output went back to low and the interrupt would be triggered again. By using a flag within the interrupt that kept track of which state the ping was in and changed every time the interrupt triggered and a free running timer that was used to record the time at each interrupt and then subtract them when the interrupt triggered and the flag changed, the time of flight of the ping could be determined, and knowing the speed of flight, the distance could be then calculated.

After setting up the interrupts, I connected the trigger and echo pins of the sensor to the digital output and interrupt pins of the Uno32 respectively (see Figure 6).



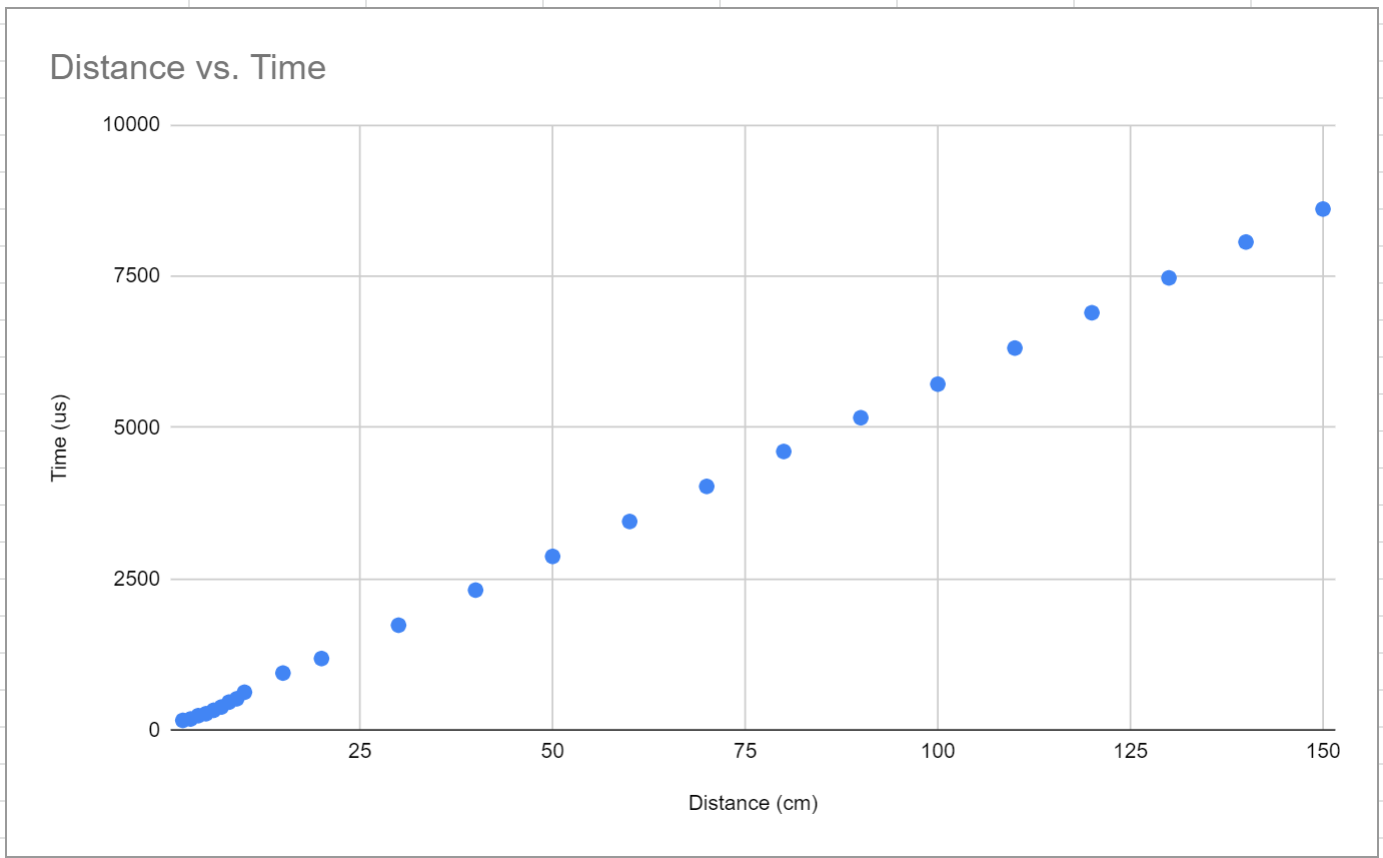
*Figure 6*

**2.2**

The next thing we had to do was to ensure that the trigger for the ping sensor was high for only 10 microseconds and would trigger every 60 milliseconds. To do so, I configured the timer that triggered an interrupt every 60 milliseconds to now alternate between going off every 60 milliseconds and 10 microseconds, which could be done by checking the state of the period match register. Initially, the timer was configured to go off after 60 milliseconds. When it went off and the interrupt was triggered, the interrupt checked that the register was set for 60 milliseconds, made the trigger go high, and then change the register so that the timer went off in 10 microseconds. When 10 microseconds went by, the timer went off, the interrupt triggered again, the interrupt checked that the register was set for 10 microseconds, and then made the trigger go back low and then set the timer back to 60 milliseconds. By doing it this way, I ensured that the ping sensor was triggered every 60 milliseconds, with the trigger lasting for 10 microseconds.

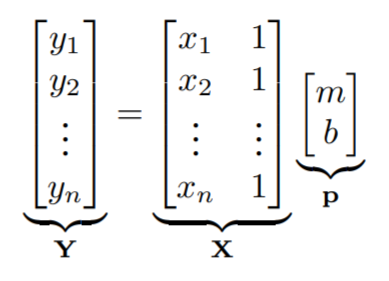
**2.3**

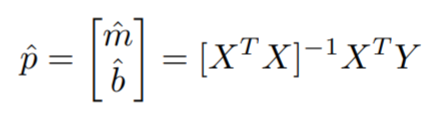
The next thing we had to do was to find a linear equation that related the time of flight of the sensor to the distance from the sensor that utilized the method of Least Squares. To do so, I took measurements of the time of flight at various distances, and added these values into Google Sheets (see Figure 7).



*Figure 7*

With distance as matrix A and time of flight as matrix B, I used the following formulas to solve for m and b in the equation for a line y = mx+b:

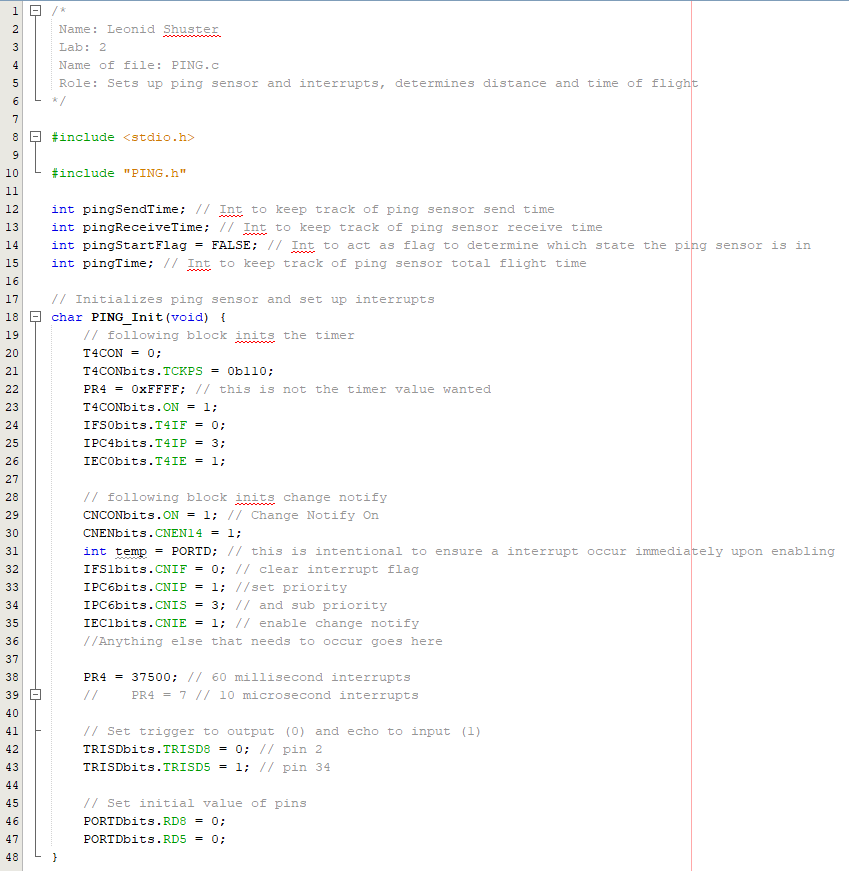




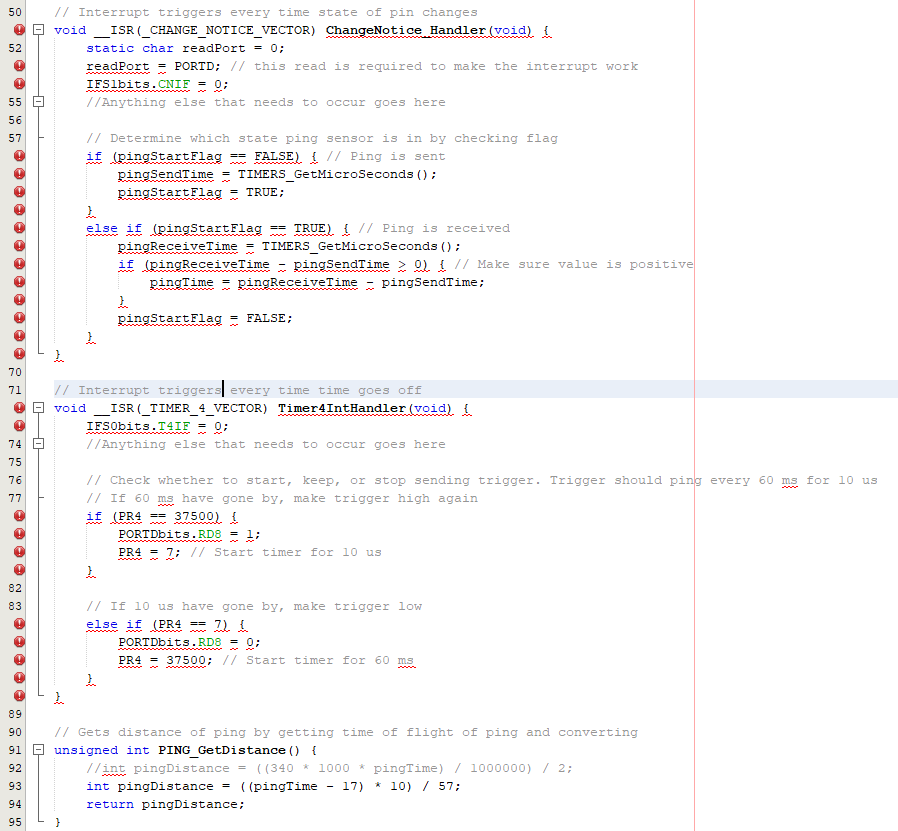
I found m to be around 57 and b to be 17.3, so the equation relating distance to time of flight turned out to be y = 57x + 17. However, I wanted distance to be dependent on time of flight, so I took the inverse of the equation and found it to be x = (y - 17) / 57. When I tested out this equation and looked at the distance found by the ping sensor, I saw that the output had a slightly smoother transition when the distance from the sensor changed.

**2.4**

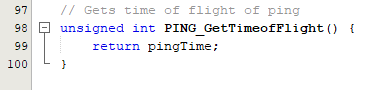
The last thing we had to do with the sensor was create a tone on a speaker based on the distance from the ping sensor. To do so, similar to previous labs, I scaled the values from the sensor to be between the minimum and maximum tones of the speaker, 0 and 1023, and sent its output through a PWM (see Figures 8.1-8.4).



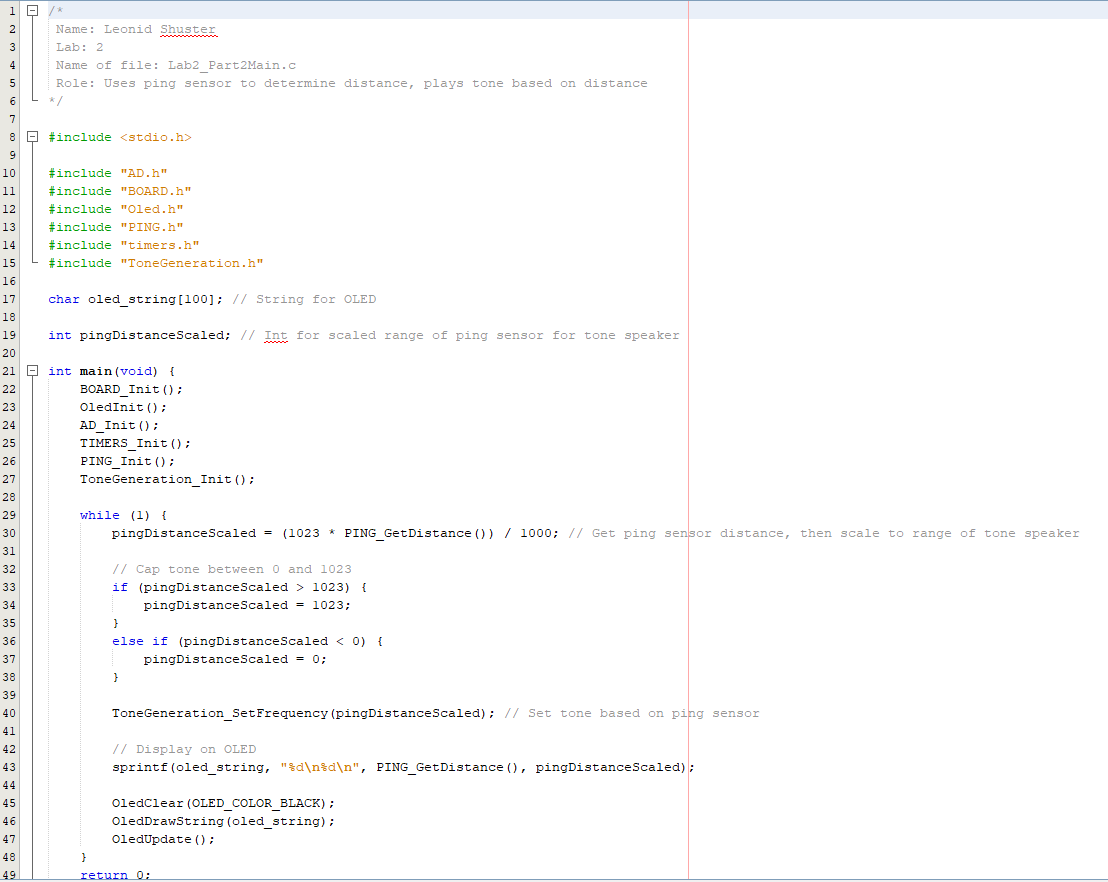
*Figure 8.1*



*Figure 8.2*



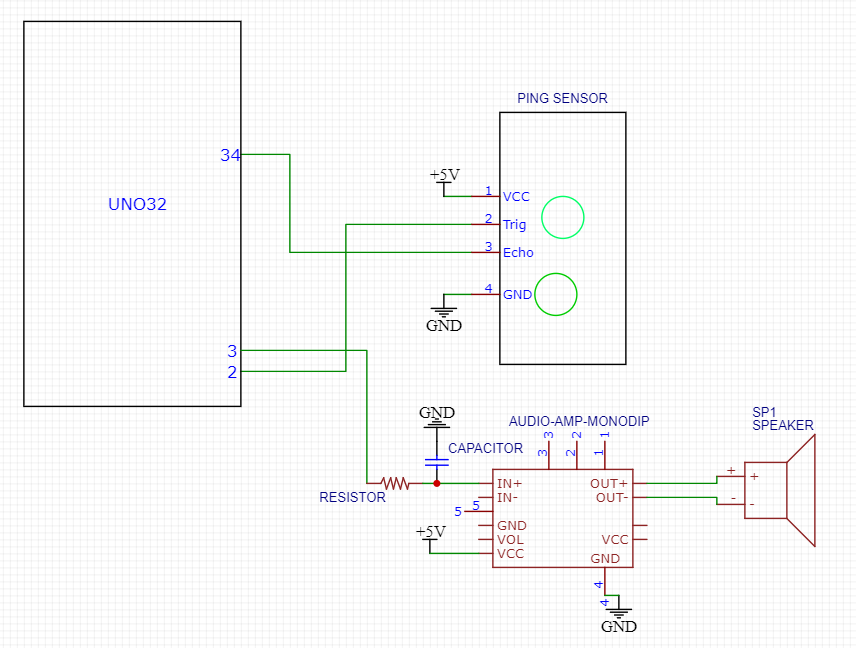
*Figure 8.3*



*Figure 8.4*

.

On the outside, I connected the tone generation PWM pin of the Uno32 to the audio amplifier, which was also connected to a speaker (see Figure 9).



*Figure 9*

**Part 3: Capacitive Touch Sensor**

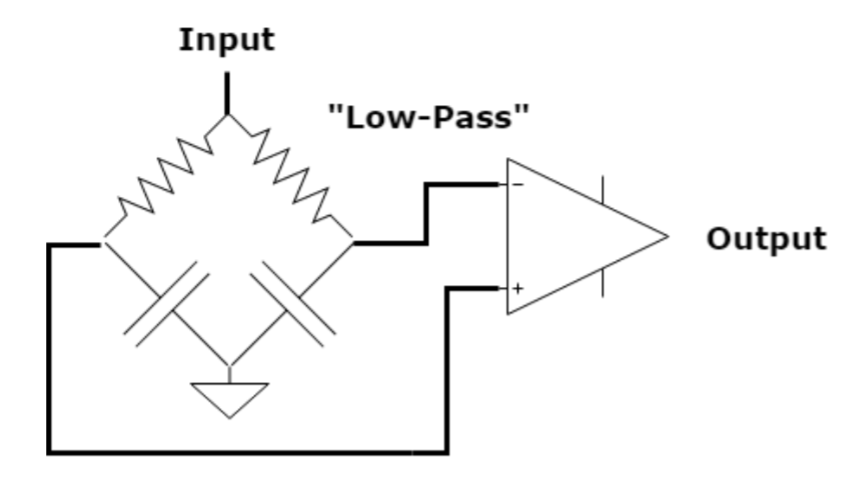
In the last part of the lab, we were told to set up a capacitive touch sensor to detect touch. The third part consisted of the following steps:

**3.1**

The first thing we had to do was measure the capacitance change in the sensor when not being touched vs. touched. To do so, I created a low-pass filter with a 300 kOhm resistor and the sensor. I then drove the circuit with a 1 kHz square wave, and observed the output on the oscilloscope. When untouched, I measured the time constant to be 20.778 microseconds, and found the capacitance to be 70.194 pF. When touched, I measured the time constant as 184.38 microseconds, and found the capacitance to be 622.884 pF. As a result, the change in rise time was 163.602 microseconds, and the change in capacitance was 552.691 pF.

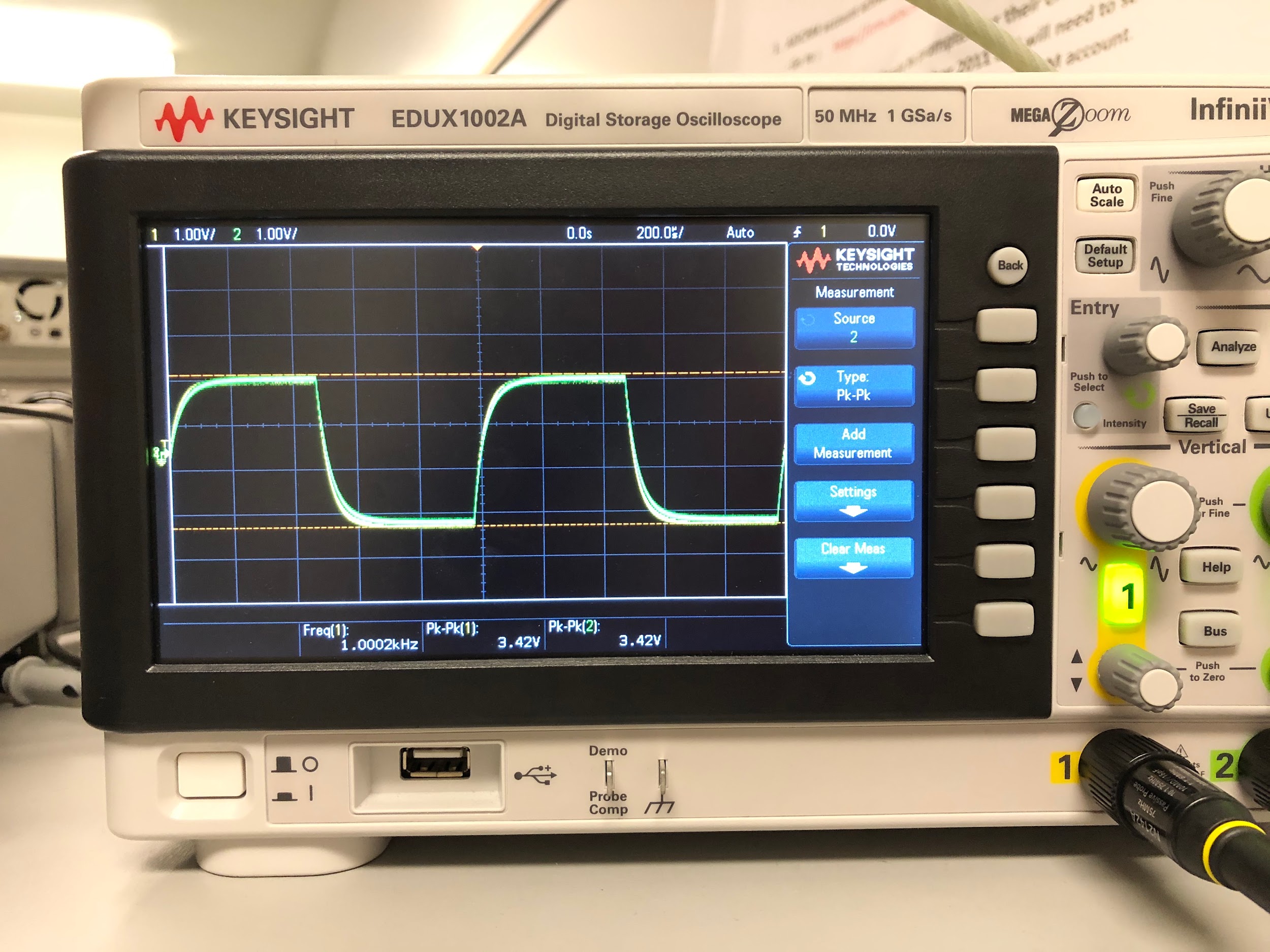
**3.2**

The next thing we had to do was integrate the capacitive sensor into a capacitive bridge and difference amp, and again observe the change in capacitance. The first circuit I created was a low-pass capacitive bridge (see Figure 10).

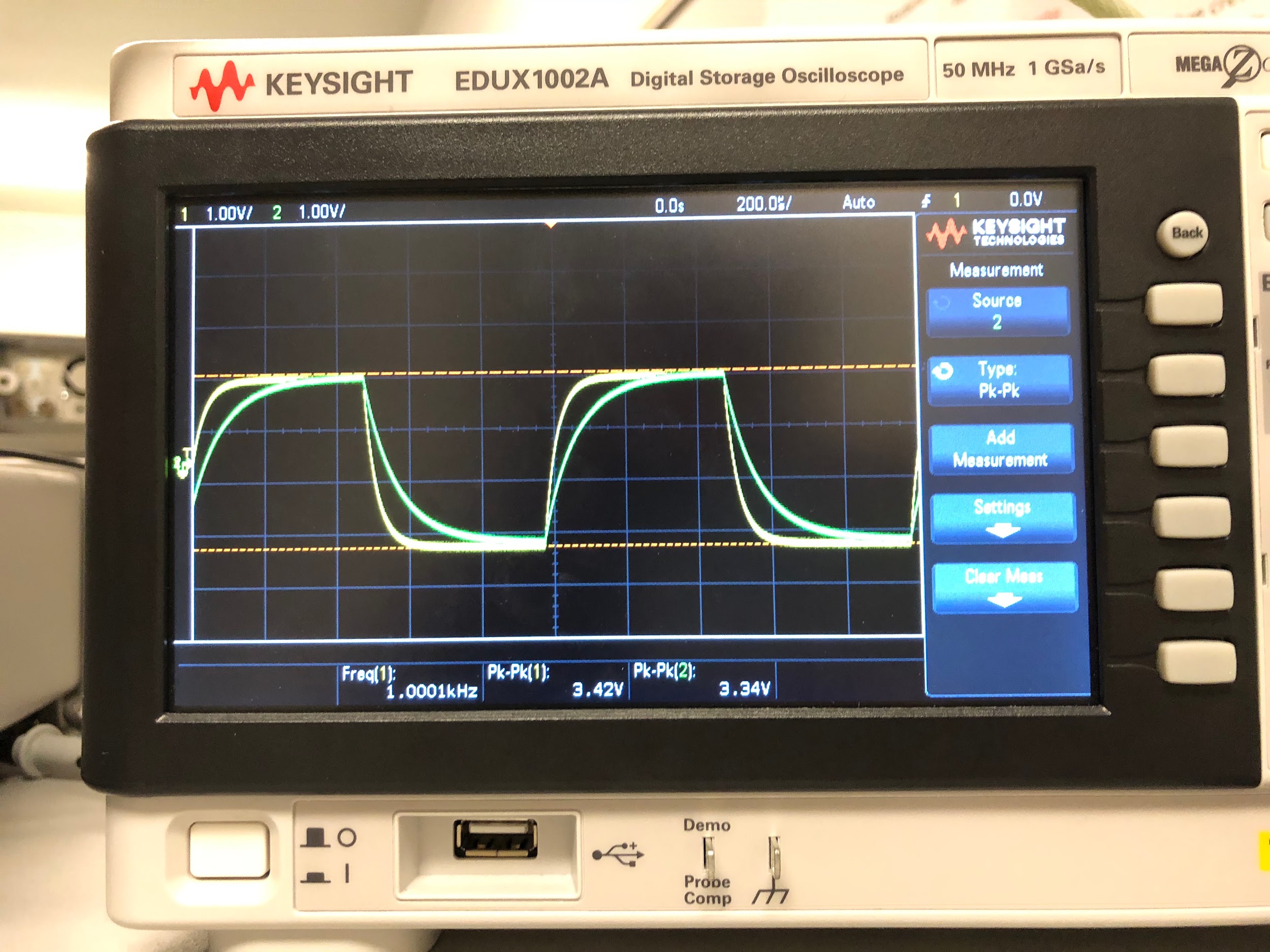


*Figure 10*

Without the op-amp, I added the touch sensor in parallel with one of the capacitors, drove the circuit with 1 kHz square wave, and hooked up both sides of the bridge to the oscilloscope and observed as the signals changed when I touched the sensor (see Figures 11 and 12).



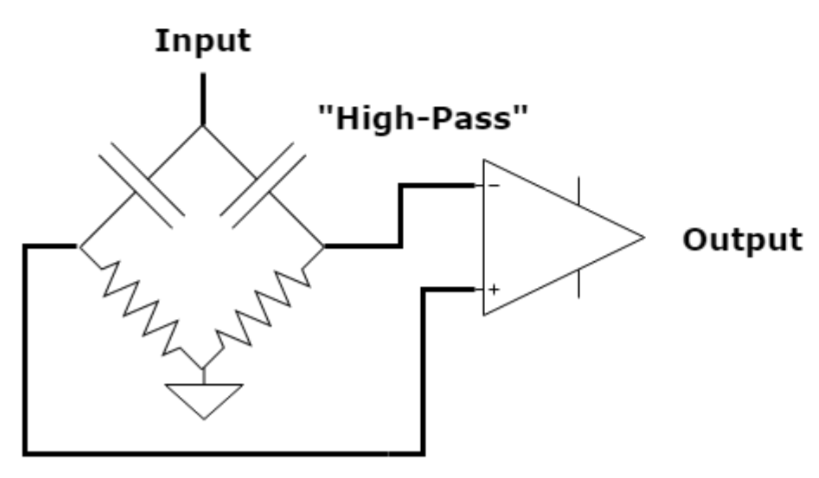
*Figure 11*



*Figure 12*

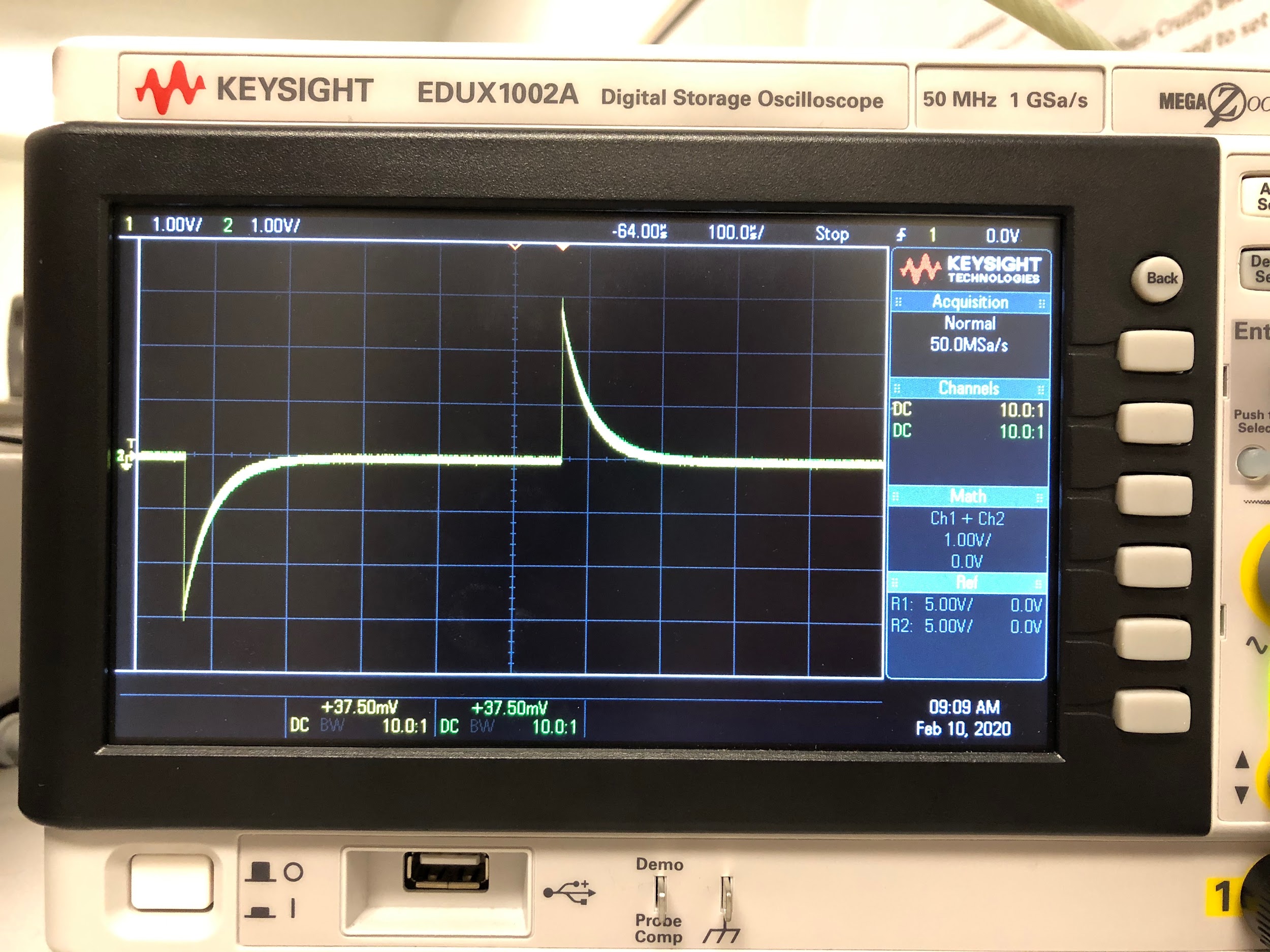
I saw that when the sensor wasn’t being touched, both sides of the bridge looked pretty much the same. When I touched the sensor, the time constant became larger, meaning the capacitance had increased, which agreed with the results I got in the previous part.

I then repeated the same process but with a high-pass capacitive bridge instead (see Figure 13).

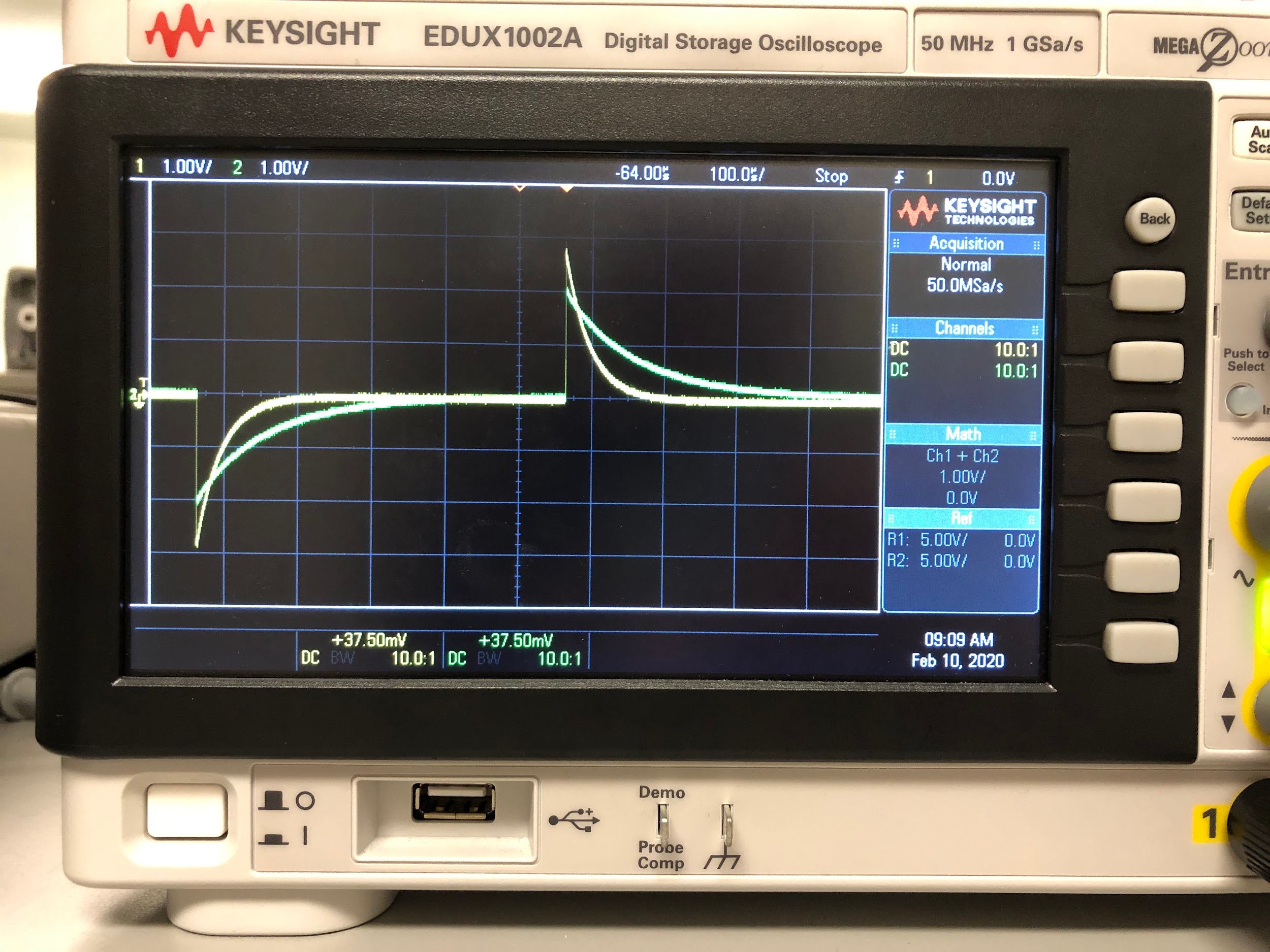


*Figure 13*

Again, without the op-amp, I added the touch sensor in parallel with one of the capacitors, drove the circuit with 1 kHz square wave, and hooked up both sides of the bridge to the oscilloscope and observed as the signals changed when I touched the sensor (see Figures 14 and 15).



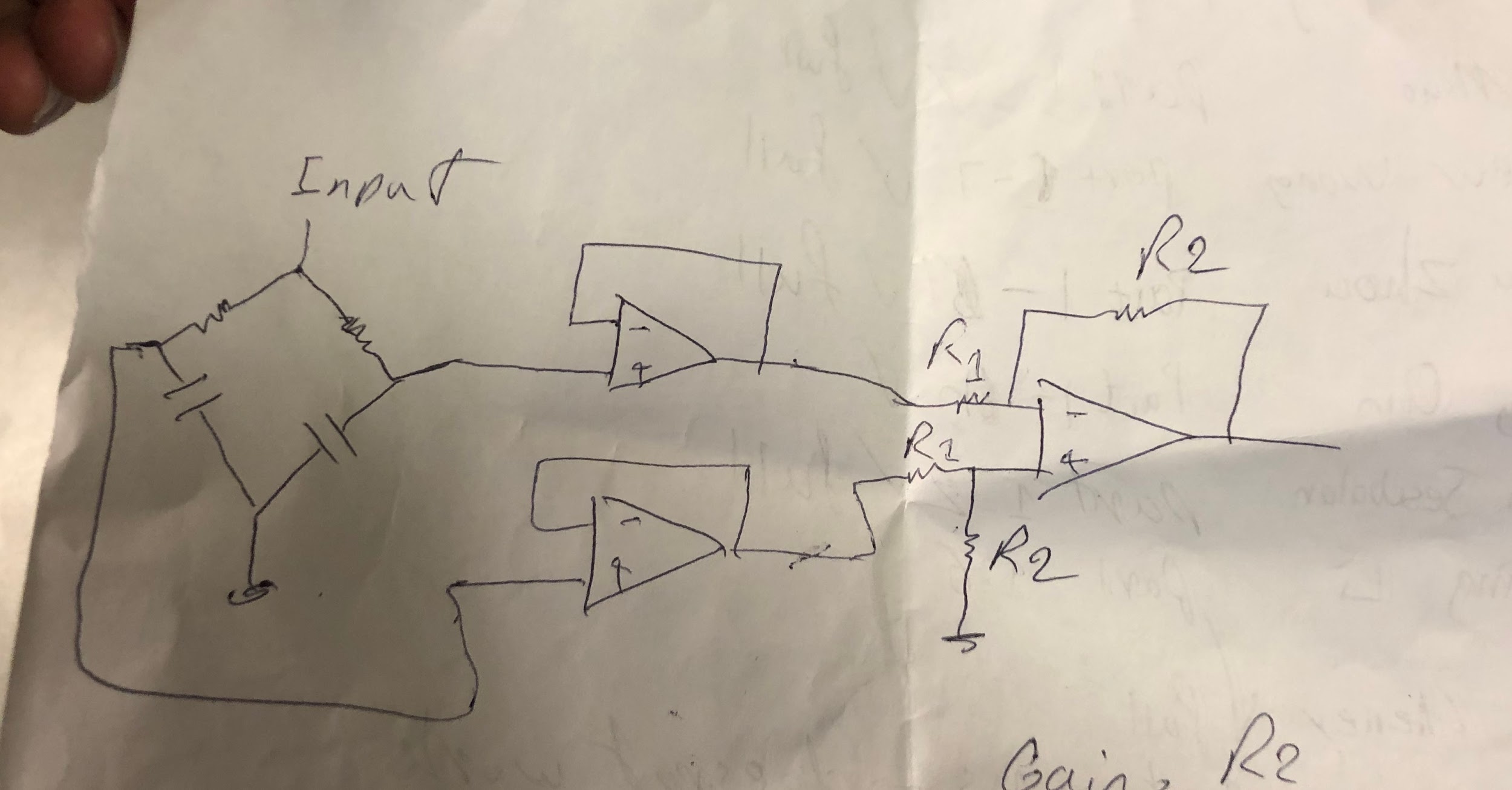
*Figure 14*



*Figure 15*

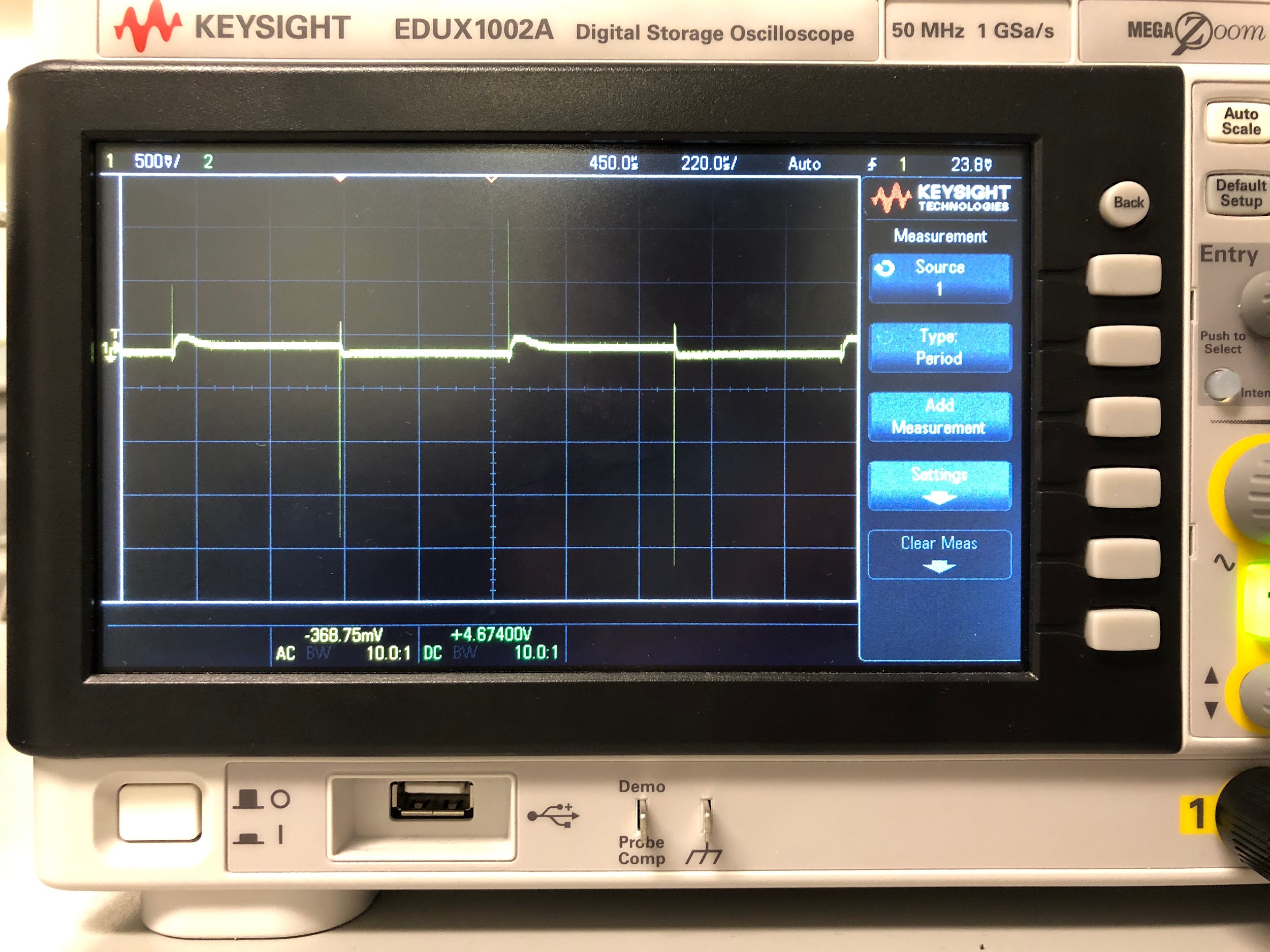
Just as before, the time constant of the signal and thus the capacitance of the sensor increased when I touched it.

Next, we had to choose either the low-pass or high-pass filter and integrate it with a difference amp. I used a low-pass filter, and created a circuit with unity gain and a follower/buffer for each side of the bridge (see Figure 16).



*Figure 16*

I then drove the circuit with 1 kHz square wave, and observed how the signal changed when I touched the sensor (see Figures 17 and 18).

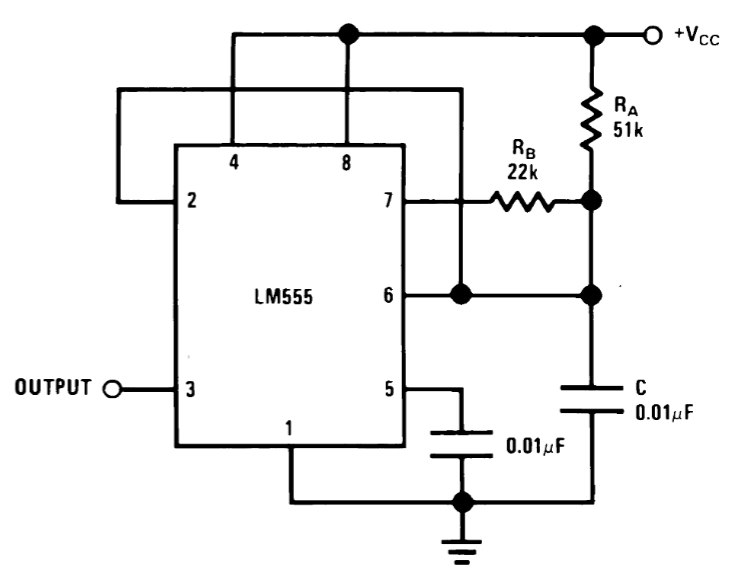


*Figure 17*



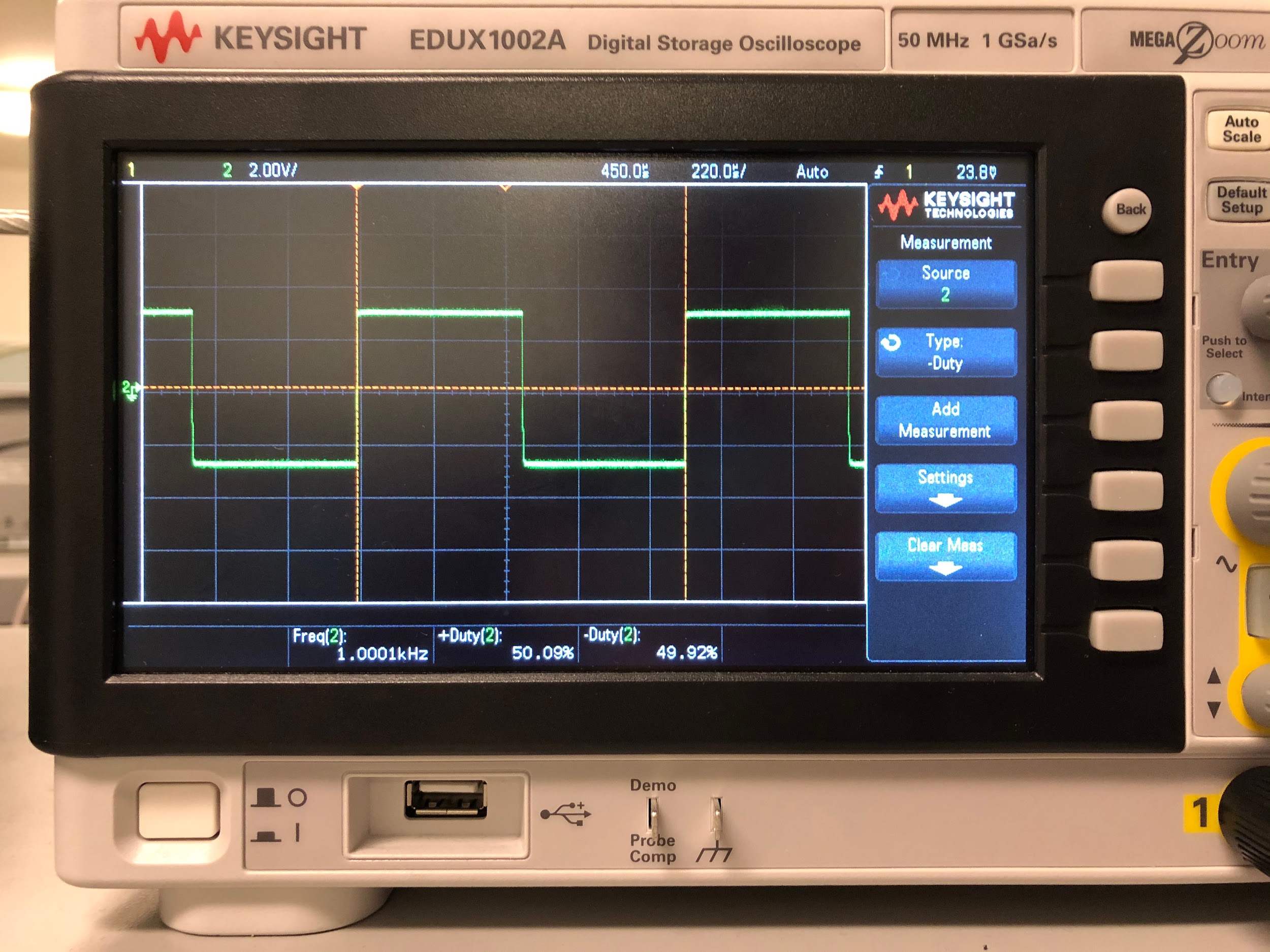
*Figure 18*

Next, we had to configure the LM555 to act as our signal generator and have a 50% duty cycle. In order to do so, I looked in the datasheet for the LM555 and found an example circuit that produced a 50% duty cycle (see Figure 19).



*Figure 19*

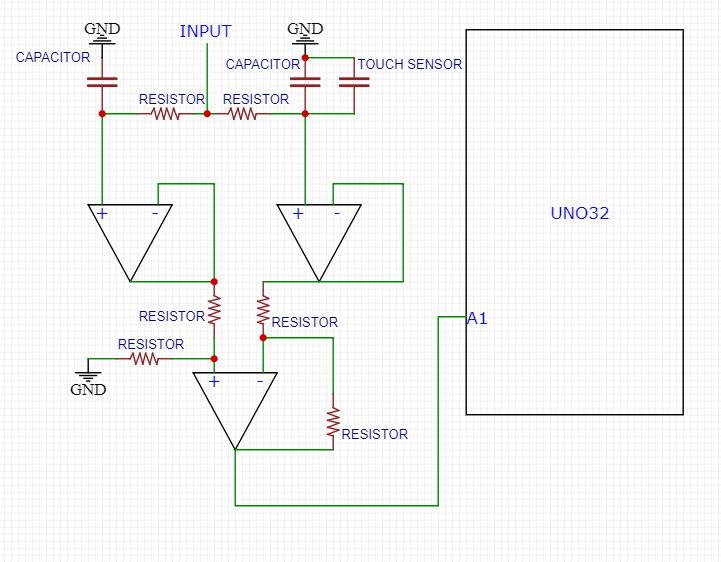
I hooked up the output of the LM555 to an oscilloscope and verified that the signal did indeed resemble a square wave with a duty cycle of 50% (see Figure 20).



*Figure 20*

Finally, we had to connect the output of the difference amp to the Uno32 and write software that detected when the sensor was touched. Because the input to the circuit was a square wave, the output of the difference amp wavered up and down, so checking for a simple threshold when touched wouldn’t work. Instead, I used a moving average of the output to detect whether the sensor was touched. When the sensor wasn’t touched, the average was lower, but when it was, the average became higher. I simply had to check when the average became above the threshold, and I was able to detect when the sensor was touched.

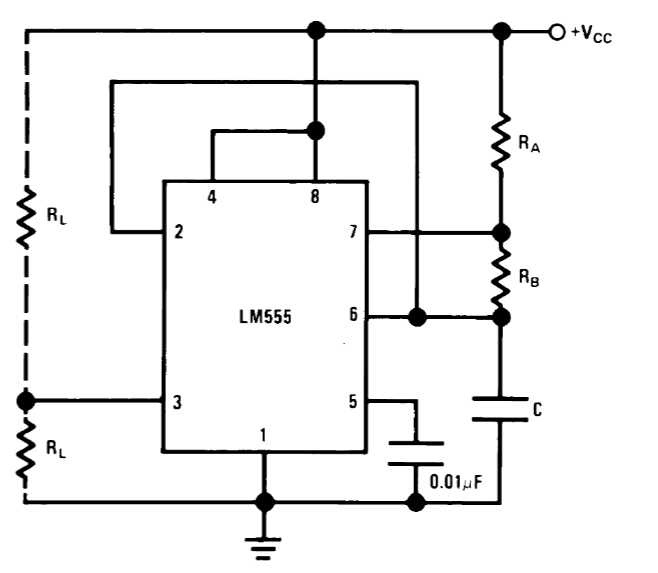
On the outside, the output from the difference amp was connected to an analog pin on the Uno32 (see Figure 21).



*Figure 21*

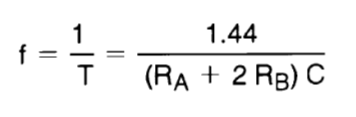
**3.3**

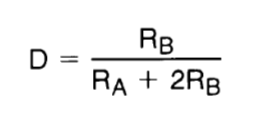
In the next part, we had to implement a relaxation oscillator using the LM555. To do so, we had to set up the LM555 to be in astable mode such that it oscillated between 1-5 kHz with a 50% duty cycle. I looked in the datasheet of the LM555 and found the diagram for astable mode (see Figure 22).



*Figure 22*

Then, in order to make sure the oscillation was between 1-5 kHz and the duty cycle was 50%, I followed these equations found in the datasheet:



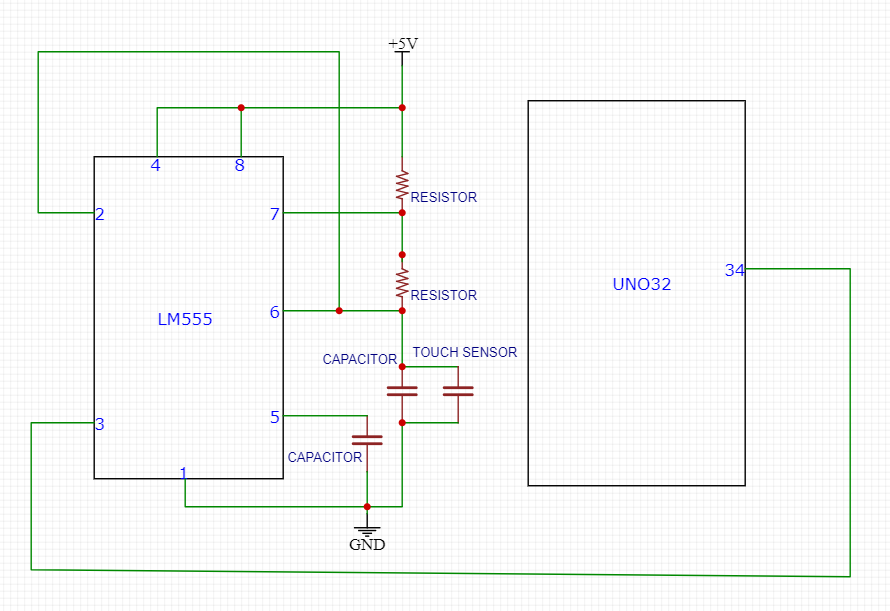


With C being a 22 pF capacitor in parallel with the touch sensor, I found Ra and Rb to be 1 kOhms and 3.6 mOhms, which should’ve given me a frequency around 5 kHz and a duty cycle around 50%. When I looked at the output from the LM555, I verified that my signal fulfilled those criteria. When the touch sensor was untouched, the output produced about 5 kHz wave, and when touched, the output became around a 1 kHz wave.

**3.4**

In the last part, we had to use the output from the LM555 circuit integrated with the touch sensor to detect whether the sensor was being pressed. To do so, we had to use an input capture interrupt that stored the value of a timer into a register when the signal changed and the interrupt triggers. By letting the interrupt trigger twice and reading the value of the timer register in each interrupt and using a flag to determine the state of the signal, the period of the signal could be determined by subtracting the two values. When the sensor was touched, the signal changed, therefore changing the period, and so touch could be detected by checking if the period was above a certain threshold. However, the signal was prone to noise, so a moving average had to be implemented to make the threshold checker more accurate. After implementing the moving average, I was able to accurately detect whether the sensor was being touched.

On the outside, the output of the LM555 was connected to the input capture interrupt pin of the Uno32 (see Figure 23).



*Figure 23*

**Conclusion**

After having gone through the lab, I feel like I fundamentally understand how to use the quadrature encoder, time based sensor, and capacitive sensor, as well as how to implement interrupts into my program. From setting up the quadrature encoder to determine position and direction of rotation in order to display a color wheel on its RGB LED in the first part, using the ping sensor to find distance from the sensor and playing a tone on a speaker in the second part, and detecting touch on a capacitive sensor in the third part, I feel confident that I can effectively use the encoder and sensors. If I were to do this lab again, I would spend more time understanding how the interrupts worked, since their initial setups were given to us.