ECE 167/L: Sensing and Sensor Technologies
Lab 2 Report

Jose Santiago

Table of Contents

Lab Introduction and Overview

This lab is an introduction to quadrature encoders, time based sensors, and capacitive sensors. The quadrature encoder used in this lab is a rotary switch with an RGB LED. The objective for this device is to implement the quadrature encoder interface (QEI) to determine the position of the encoder and drive the color of the RGB LED as appropriate. The time based PING sensor will be used to measure the distance of the sensor to various objects. The PING sensor will have to be linearized using Least Squares. The capacitive touch sensor will be used to experiment with several techniques to measure change in capacitance. Every sensor will need modular non-blocking code developed for the sensor, enabling the user to interact with them and read them.

The hardware necessary to complete this lab includes:

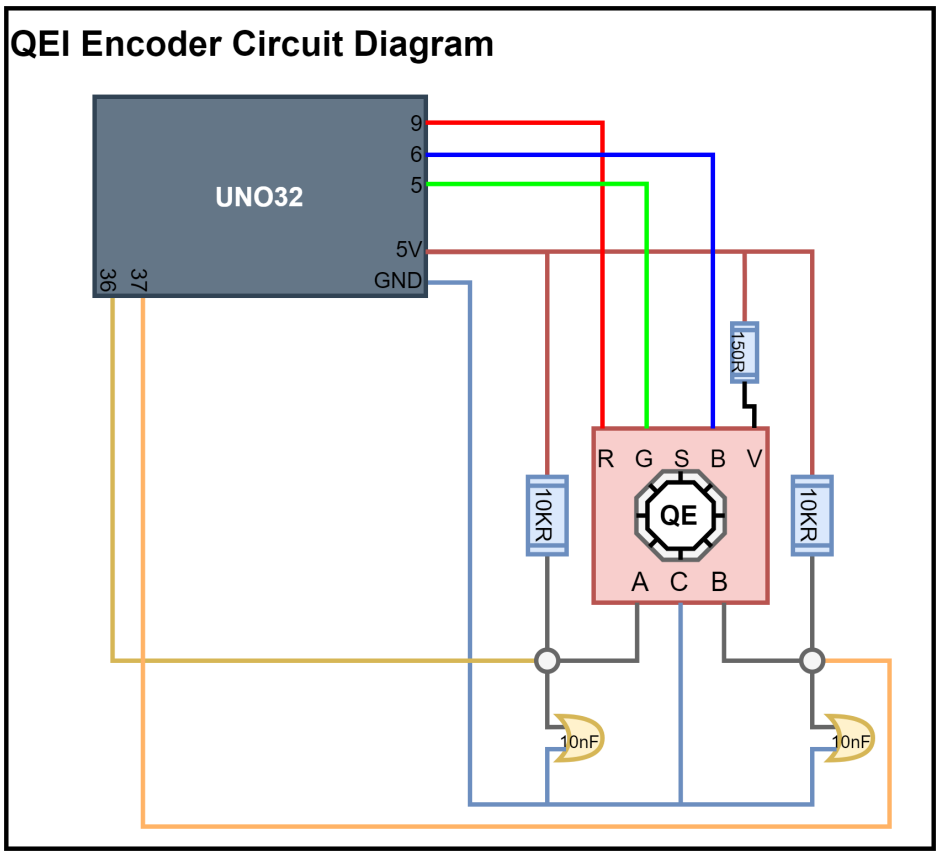
- UNO32
- Speaker
- Audio Amp (with potentiometer for volume control)
- Breadboard
- Rotary encoder
- PING sensor
- Capacitive touch board
- LM555 timer chip
- MCP6004 quad OpAmp chip
- Assorted capacitors and resistors

Quadrature Encoder

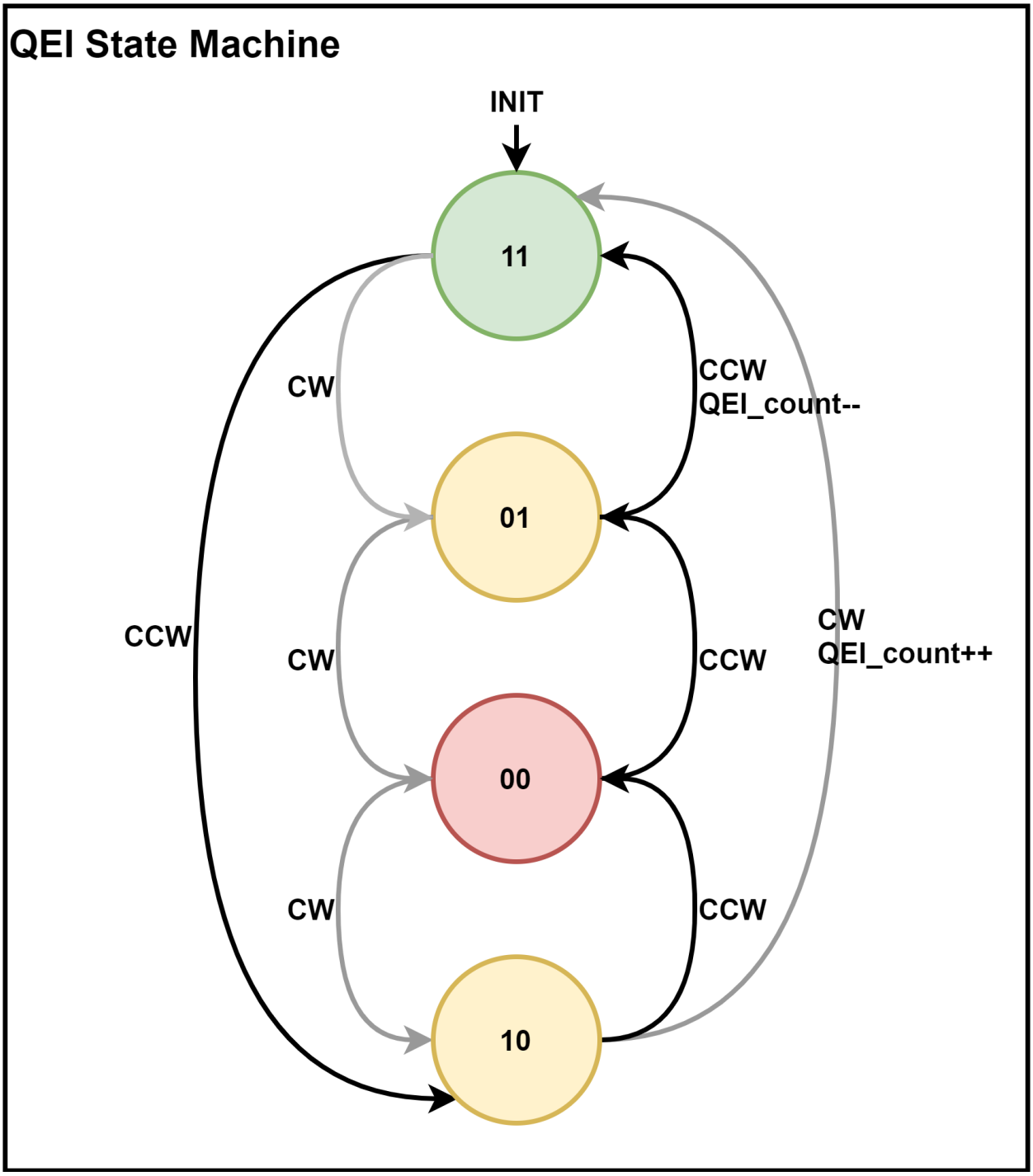
A quadrature encoder has a signal that consists of two square waves that are 90 degrees out of phase with each other. By comparing the state of one of the square waves to the other, we can determine the direction the encoder is turning as well as accumulating a count or angle. The two square waves of the encoder are usually designated as “A” and “B.”

The first step for implementing this device is to wire it on a breadboard and connect it to the UNO32. Following the circuit diagram found in page 4 of the datasheet we can implement the proper circuit to begin working with the quadrature encoder. The circuit diagram can be seen below. The components necessary to build the circuit are: the UNO 32, two 10 kOhm resistors, one 150 Ohm resistor, two 10 nanoFarad capacitors, and several wires. The two 10 kOhm resistors and two 10 nanoFarad capacitors are used for the encoder ports, A and B. The 150 Ohm resistor is meant to prevent the RGB LED from burning out.

After setting up the circuit the next step is to start reading the A and B ports of the Quadrature Encoder through pins 36 and 37 of the UNO 32. Both of these pins are configured in the QEI_Init() function to trigger the Change Notify peripheral set up. The Change Notification function of the I/O ports allows the PIC32 devices to generate interrupt requests to the processor in response to a change-of-state on our selected input pins, 36 and 37. Using this peripheral we can keep track of when the A and B pins change in value. A state machine can be



used to determine the rotational direction of the Quadrature Encoder. We can determine the direction of the Quadrature encoder via a state machine. We can use states that represent the state of the A and B signals. There are four different states possible representing the A and B signal states. The states for A and B can be represented as bits: 11, 01, 10, and 00. When the encoder is turned clockwise, the A and B signals follow the pattern: 11, 01, 00, 10, 11. When the encoder is turned counterclockwise, the A and B signals follow the pattern: 11, 10, 00, 01, 11. Knowing this, we can create a state machine that transitions every time the Change Notice Handler is triggered. Depending on the current state of the state machine, and the current value of ports A and B of the encoder, we can determine the next state of our state machine. The initial state for the state machine is when both signals are high, 11. There are two possible A and B values that can be read from this state, 01 for clockwise rotation, and 10 for counterclockwise rotation. From state 01 we can go to state 00 for clockwise rotation or state 11 for counterclockwise rotation. From state 00 we can go to state 10 for clockwise rotation or state 01 for counterclockwise rotation. From state 10 we can go to state 11 for clockwise rotation or state 00 for counterclockwise rotation. These transitions describe all possible values that can be read from the Quadrature Encoder. A visual representation of the state machine can be seen below.



An important point to mention is since we want to accumulate a count for the different rotations of the quadrature encoder, we should keep a module variable QEI_count, that keeps track of our rotational count. Clockwise rotations increment the count variable while counterclockwise rotations decrement the count variable. We want to make sure we increment or decrement whenever we complete a full transition in our state machine in the clockwise or counterclockwise direction. Therefore we want to increment the QEI_count variable when we

transition into the 11 state from the 10 state, and we want to decrement the QEI_count variable when we transition into the 11 state from the 01 state.

Now that the structure for our program has been described, we can begin implementing it in code. The QEI.h file includes some starter code along with the prototypes for the functions necessary for our implementation. We have two module variables defined in QEI.c. One of them is the previously mentioned QEI_count variable which keeps track of the count of our Quadrature Encoder, the other is the QSTATE variable S. The QSTATE variable, S, is a variable which holds one of the states for our state machine. The QSTATE type is an enum type which has members of the QEI state machine, 00, 01, 10, and 11.

The first function we need to implement is the QEI_Init() function. This function performs all the proper initialization so we can keep track of our Quadrature Encoder rotations. A lot of the code for the initialization function has been written, but we need to define our initial state, the initial QEI_count value, and return SUCCESS as defined in the BOARD.h file.We set our state variable S to the 11 state within QEI_Init() and we set the QEI_count variable to 0. The QEI_GetPosition() function returns the QEI_count variable. The QEI_ResetPosition() function sets the QEI_count variable to zero, resets the state variable S to 11, and clears the interrupt flag. There is a QEI_SM() function which is a switch case statement that implements our state machine as described earlier. The argument of the switch is the state variable S. There are four cases, one for each state of the state machine. Within the states we check the value of the A and B ports to determine what state to change into and whether or not to increment or decrement the QEI_count variable as depicted in our state machine diagram. The final order of business for this module is to implement the Change Notice interrupt service routine. We already have starter code within this ISR. All we need to do in this ISR is call the QEI state machine to keep track of the state of our state machine.

After implementing the Quadrature Encoder Interface, we can use the angle of the encoder in degrees to determine the color of the RGB LED. To do this we must first wire the RGB LED component of the Quadrature Encoder. This is simple to do, we just need to connect the RGB pins of the Quadrature encoder to different PWM pins on the UNO32. In this implementation we use pin 9 for red, pin 5 for green, and pin 6 for blue. Then we connect the voltage pin to 5V with a 150 Ohm resistor so the LED does not burn out.

After wiring the RGB LED, we can implement the code to map our degrees to different RGB color combinations to imitate the color wheel. First we need to include the pwm.h header file to enable pwm functions for our pins to be able to light our LED's. We also want to include our QEI.h header to determine the Quadrature Encoder's position. The Oled.h file is included so we can use the OLED screen for debugging purposes. The BOARD.h file is included as always to be able to interface with the UNO 32. To make our code easier to weird we can #define the PWM pins to their corresponding pin numbers on the UNO 32. Therefore PWM_PORTY04 is aliased as PIN_9, PWM_PORTY12 is aliased as PIN_5, and PWM_PORTY10 is aliased as PIN_3.

Within the main program we initialize all the different modules included, BOARD, OLED, PWM, and QEI. We then add the PWM pins using PWM_AddPins() so we can set the duty cycles of our output PWM pins. Since our RGB LED is a common anode LED, the larger the PWM duty cycle, the less current would run through our LED, meaning the light from the respective LED would be less. In this configuration the UNO 32 pin serves as a sink to the

current supplied by the voltage input to the common anode pin. To set the duty cycle of the PWM pins we use the pwm.h function PWM_SetDutyCycle(). This function takes a pin whose duty cycle we wish to set, and an unsigned int depicting the duty cycle. The duty cycle can range from 0 to 1000. Reading up on the RGB color wheel, you can find different RGB color combinations for the colors in the color wheel. Since we know one full revolution of the encoder happens with 24 tics of the encoder in one direction, we can deduce that one tic of the encoder corresponds to a change in angular position of 15 degrees. Within our infinite while loop, we want to store the Quadrature Encoder position in a variable, q, to be able to convert it to degrees. To convert our q variable into degrees, we take the absolute value of the variable q, multiply it by 15, and take the modulus of 360. We take the absolute value because we don't want a negative degree. We multiply by 15 because we know every tick of the encoder is 15 degrees. We then take the modulo of 360 because we want to start the angle at 0 again after a full 360 degree revolution.

To make things simpler we define duty cycle variables for the three colors, red (R_DC), green (G_DC) and blue (B_DC). Then we can cascade several if-else-if statements to be able to assign the duty cycle variables to the R, G, B pins depending on the 24 different angles in one revolution of the Quadrature Encoder. Below the cascaded if-else-if statements we use the duty cycle variables to set the duty cycle of the RGB pins with the PWM_SetDutyCycle() function. Underneath this we print out pertinent information to the CoolTerm terminal and OLED screen. We print the RGB duty cycle, degrees and count given the current degree of rotation of the Quadrature Encoder.

PING Sensor

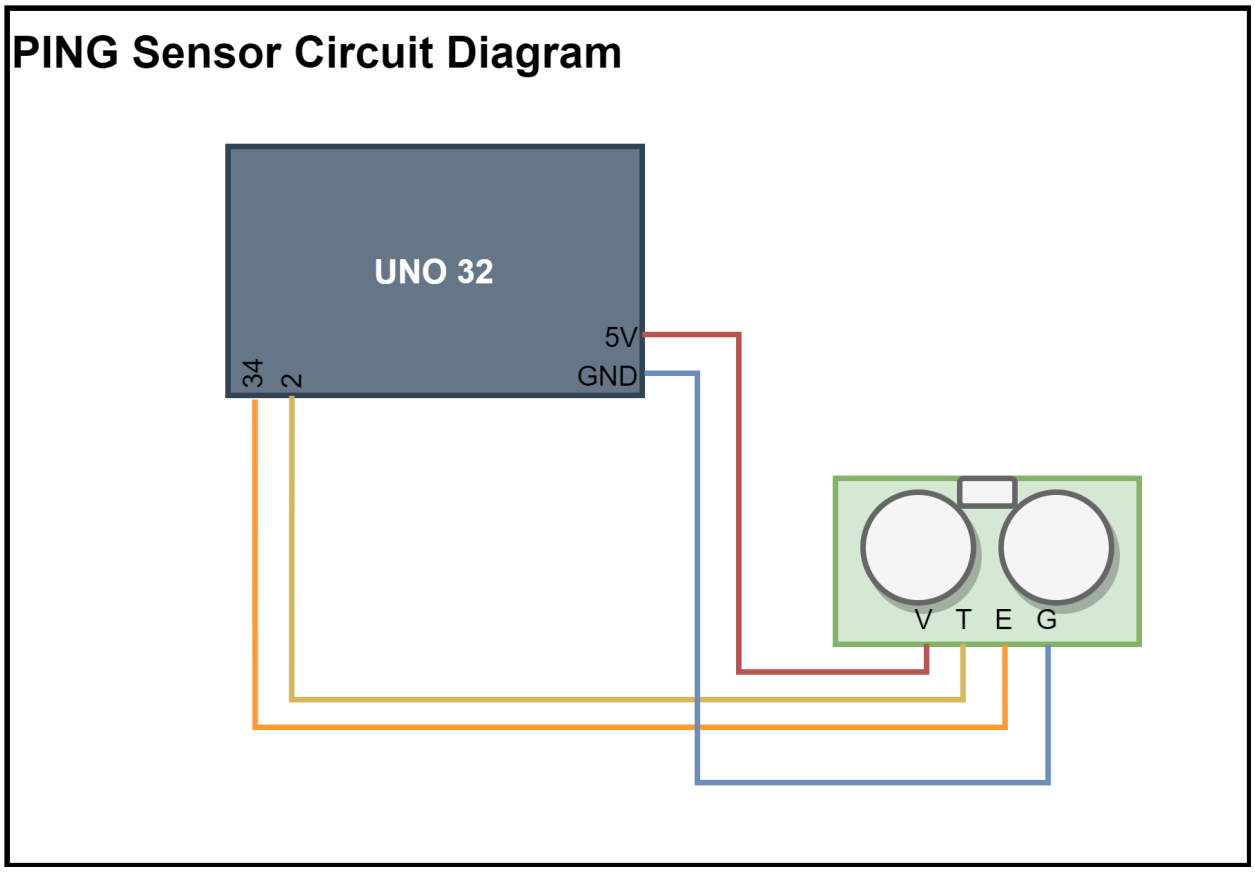
The next sensor used in this lab is a time-of-flight ranging sensor called a "ping" sensor. This sensor uses ultrasound (~40 kHz) to measure the time of flight between an ultrasound pulse emission and the return echo that bounces back from an object in the sensor's direct path. The time of flight speed of the ping from this sensor is given as 340 m/s in the datasheet. Since we are measuring the time it takes the ultrasound ping to travel to and from an object, we will want to halve the time since we are only interested in the time it takes the ping to reach an object. Therefore we can calculate the distance of an object from the ping sensor using the formula d = v * t * 0.5. Since it is specified in the lab manual that we should not use floating point math to halve the distance the ultrasonic ping travels, we will be shifting the bits of our product to the right one time, which is the same as dividing by two.

Before implementing our software for this device we will want to set up our circuit. This circuit is relatively simple. The hardware components necessary for the circuit are the PING sensor, the UNO 32, and some wires. The circuit diagram for this sensor can be found below.

The data sheet tells us that to use the sensor, we need to activate the trigger pin for 10 us, followed by a wait period of 60 ms to allow the sensor to receive an echo. We can implement this using a state machine. The two states of interest are a PULSE state, and a WAIT state. In the PULSE state we set the trigger pin of the PING sensor high for 10 us. In the WAIT state we wait to receive an echo from the echo pinfor 60 ms.

To keep track of the time in our states we will use the timer peripheral, TMR4. This timer peripheral triggers an interrupt when the timer reaches to max value and rolls over. TMR4 is set

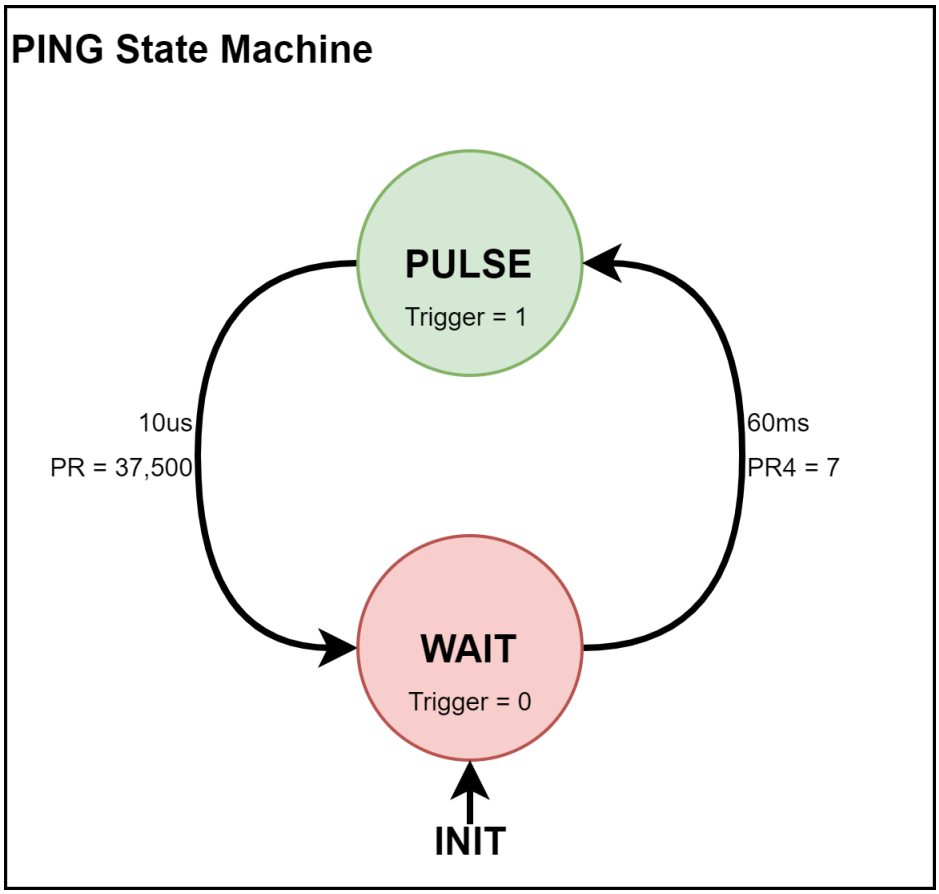
up with a prescaler of 64:1. Which means the timer increments once every 64 Peripheral Bus Clock Cycles. Using the fact that the PBCLK frequency is 40MHz, we can find the frequency of TMR4, which is 625,000 Hz. After this we can use proportional math to find the cycles necessary for 10 us and 60 ms to elapse. The desired cycles are found as 6.25 cycles for 10 us and 37,500 cycles for 60 ms. We round up to 7 cycles for 10 us. These values are the ones we will set our TMR4 maximum count values to. To do this we write to the timer's period register, PR4. Knowing this, we will have to add a line of code in our state machine to set the PR4 when we transition out of one state and into another.



Another peripheral we will use in this implementation is the Change Notice peripheral. We will use this interrupt to keep track of the state of the echo pin of the PING sensor. When the echo pin receives an input it will trigger an interrupt. This is important because we will want to keep track of the time when an echo is received using the TIMERS_GetMicroSeconds() function from the timers.h library. We will use this time in combination with a recording of the time when the ping was sent in the PING state to be able to calculate the time of flight of the ping.

Now that we understand all components necessary to get the PING sensor working, we can describe how we implement it in our code. There are four unsigned int variables we want to keep track of. We want time variables, t1, t2, and td to keep track of different times. Variable t1 keeps track of the time the PING sensor triggers a pulse. Variable t2 keeps track of the time the PING sensor receives an echo. Variable td stores the time duration of a pulse to an echo (t2 -

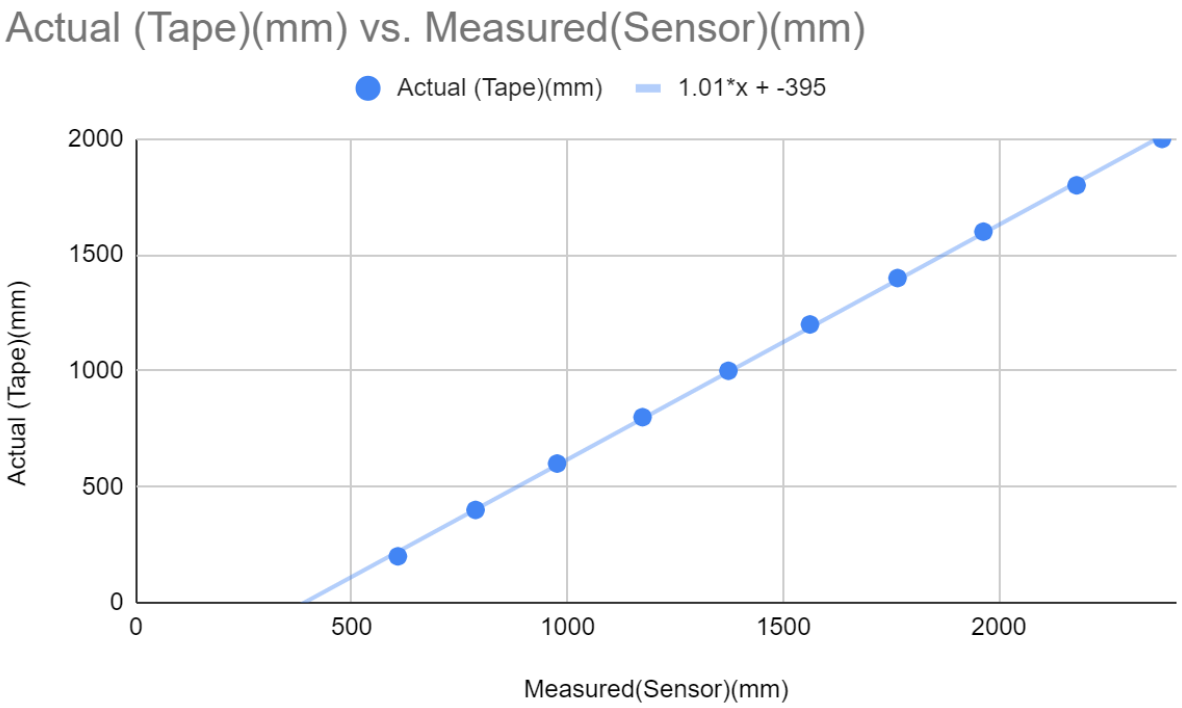
t1). The last unsigned int variable is the distance variable d. This is where we will store the distance converted from the time of flight of our ping. Another variable of importance to us is a TSTATE S variable. TSTATE is an enum type which holds both states of our state machine, WAIT and PULSE. We will use pin 2 of the UNO 32 as an output to set the trigger pin high or low. We can set this pin as an output within our PING_Init() function with the command TRISDbits.TRISD8 = 0. To make our code more easily legible, we can #define the command that references pin 2 as T, #define T PORTDbits.RD8. Now whenever we want to write to our trigger pin all we have to type is T = x (x = 0 or x = 1). Within our PING_Init() function we will also want to initialize the period register of TMR4 to 60 ms and set the initial TSTATE S to WAIT. We want our trigger pin low here too (T = 0). We will also want to initialize our timer.h functions (TIMERS_Init()) and return SUCCESS. All of the work in the program will be done within our state machine. We can define a function to implement our state machine, PING_SM(). We define this function similar to the QEI_SM() state machine function of the previous section. We have a switch-case statement, where the input to the switch statement is our TSTATE variable S. The cases within the PING_SM() function are PING and WAIT. We will be calling the PING_SM() whenever TMR4 reaches the period register value, because that is when the time necessary for either state (WAIT or PULSE) has been elapsed. The flow of the state machine can be seen below in the PING State Machine Diagram.



Within our PING_GetTimeofFlight() function we subtract the time the pulse is triggered (t1) from the time an echo is received (t2) to get the time of flight (td). We return td within this

function. Within the PING_GetDistance() function we convert time of flight to distance. We can use the equation d = (((td) * (340))/1000) >> 1. We can then return the d variable as distance in mm. Within the Change Notice ISR we store the time an echo is received into variable t2, since that is when the ISR is triggered. Within the Timer 4 ISR we call the PIN_SM() since we want to change state whenever our TMR4 value reaches the current period register value. Within our PULSE case we can store the time the ping is sent into variable t1.

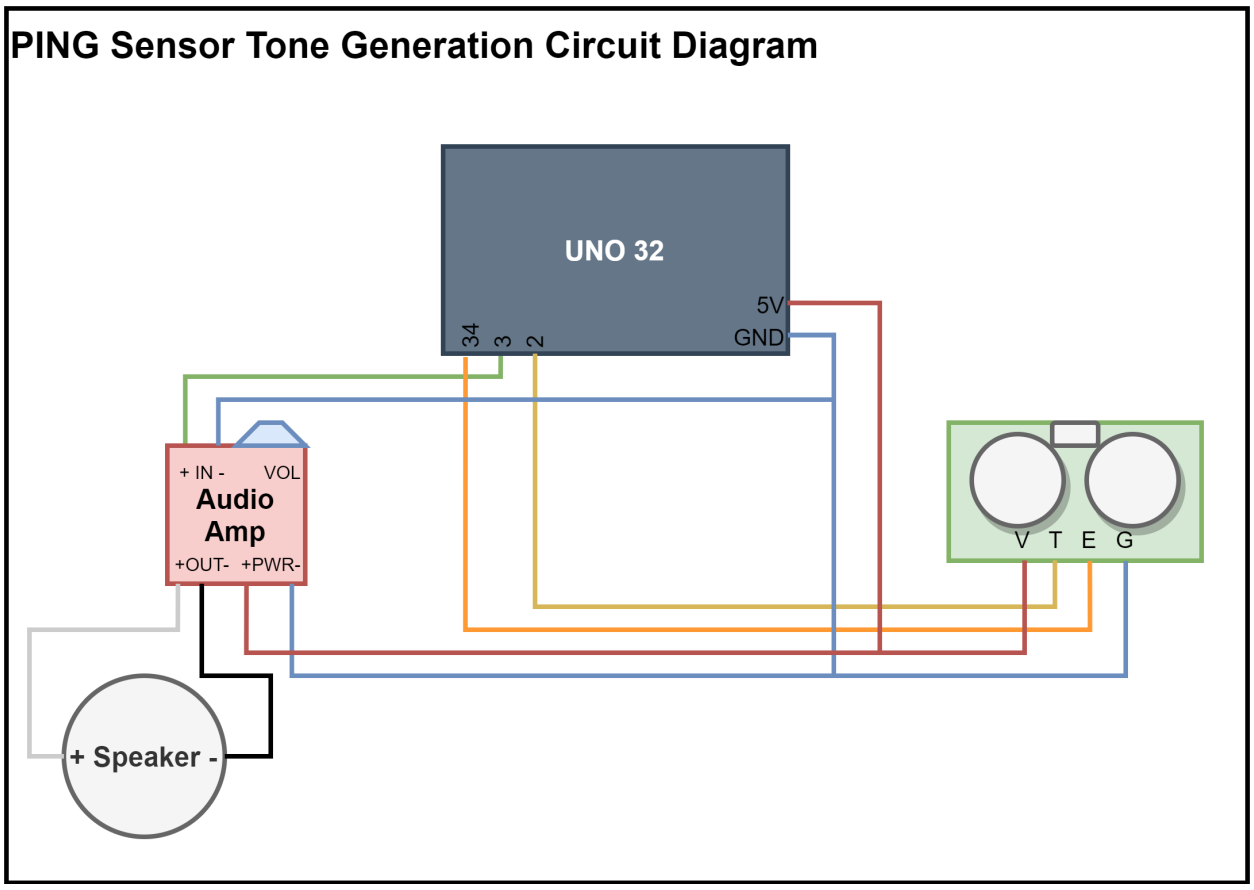
In order to increase the accuracy of our PING sensor we can implement a Least Squares linearization. To do this we need to record 10 different measurements according to the PING sensor and compare them to actual measurements away from the object with a tape measure. Using Google Sheets we can find a line that converts the expected values using the PING sensor to the actual values found from our measurements with a tape measure. Below is the line plot gathered from the given data. From this data we find the equation for our linearization



(d = (1.01 * d) - 395). We now input the data measured by our PING sensor into the equation to find the true measurement of the object from the PING sensor.

Now that the PING sensor module has been implemented we can use it to generate different tone frequencies from our speaker. Tone generation is the same as it was for the previous labs. We will add the Speaker and Audio Amp PCB components for this portion of the lab. The updated circuit diagram for this part is depicted below.

After updating our circuit to implement the tone generation, we have to write a main file to implement all these different components together. Within our main we want to initialize our BOARD, as always, along with the OLED, PING, TIMERS, and ToneGeneration modules. We will have variable, t ,to store time. We store time at the top of our infinite while loop. Then we store the time of flight and distance in variables tf and d. We will use the distance variable to



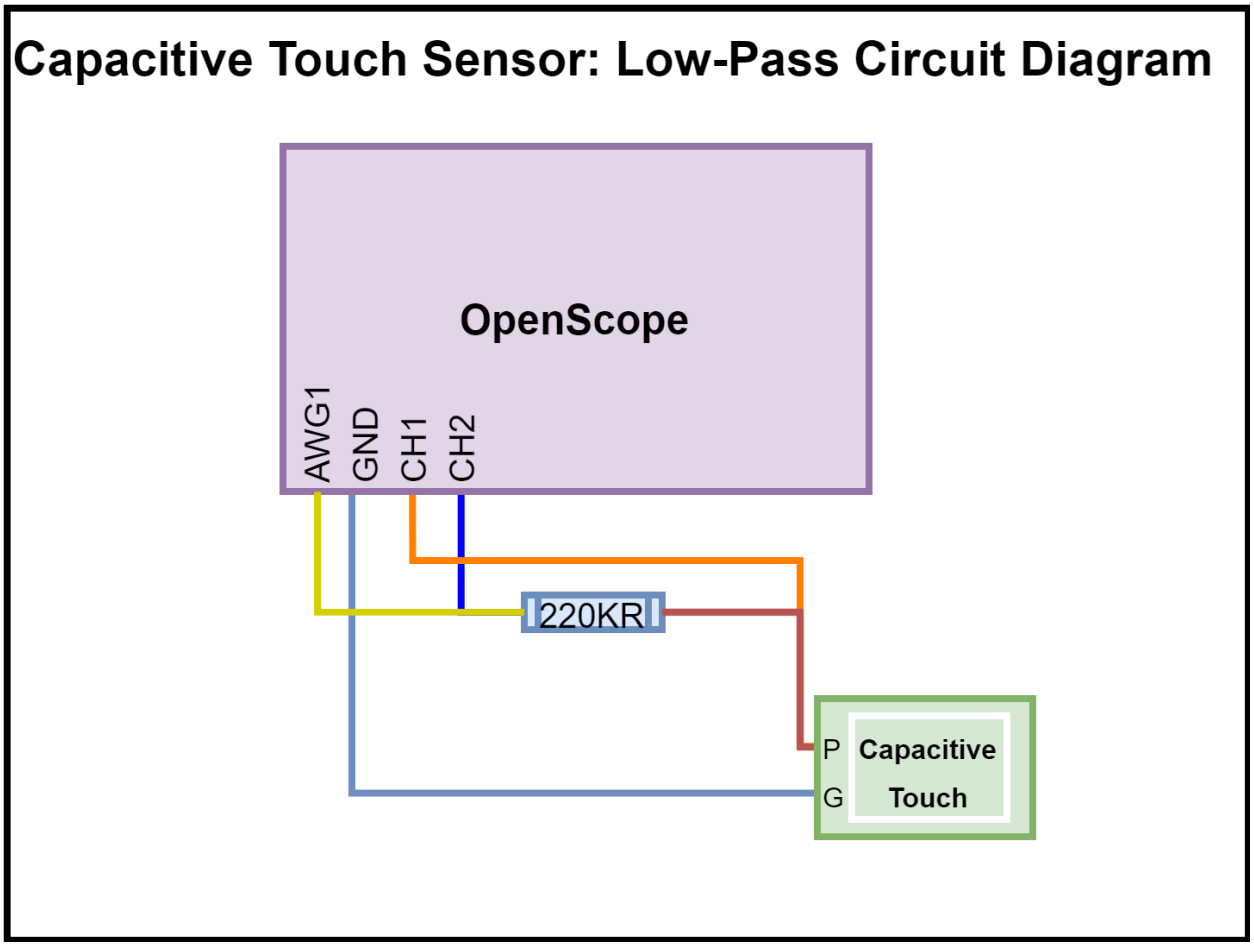
convert to the frequency we want to output. The maximum distance used in this implementation is 2000 mm. Any distance above this is ignored. We can do this with a simple if-else statement. Afterwards we can filter any noise from our PING sensor. We will do this with both distance and frequency. We will have a previous distance variable d_p to keep track of the previous distance. We will subtract this from the current distance and take the absolute value. If this value is less than 10, we don't want to track the change in distance. This means any change in distance less than one cm is not tracked in our implementation. Then we convert distance variable d into frequency. Since our ToneGeneration module goes from 1 to 1000 Hz, and we have set our distance maximum as 2000 mm, we can use a simple equation to convert distance to frequency, h = ((d*999)+1)/2000. Then we want to make sure our frequency is not above our maximum, 1000, or below our minimum 1 by checking with simple if statements and setting them to 1 or 1000 if the frequency is below or above 1 or 1000 respectively. Then we will check for a change in frequency greater than or equal to 10 to update our previous frequency. We will do this with an if statement and simple math, by subtracting the previous frequency from the current frequency and taking the absolute value. Then we can do further software filtering by filling in our frequency buffer of size 50 and averaging all the values from the 50 previous sampled frequencies. The output average is then checked to verify that the current average frequency has changed by more than 4 Hz to enable a change in frequency. Then we set the frequency with the function ToneGeneration_SetFrequency(). Afterwards we print our debugging information to the CoolTerm terminal and the OLED. This information includes the time of flight,

distance, and frequency. At the bottom of our while loop we want a wait loop to wait a tenth of a second. With this our implementation for the PING sensor to ToneGeneration is complete.

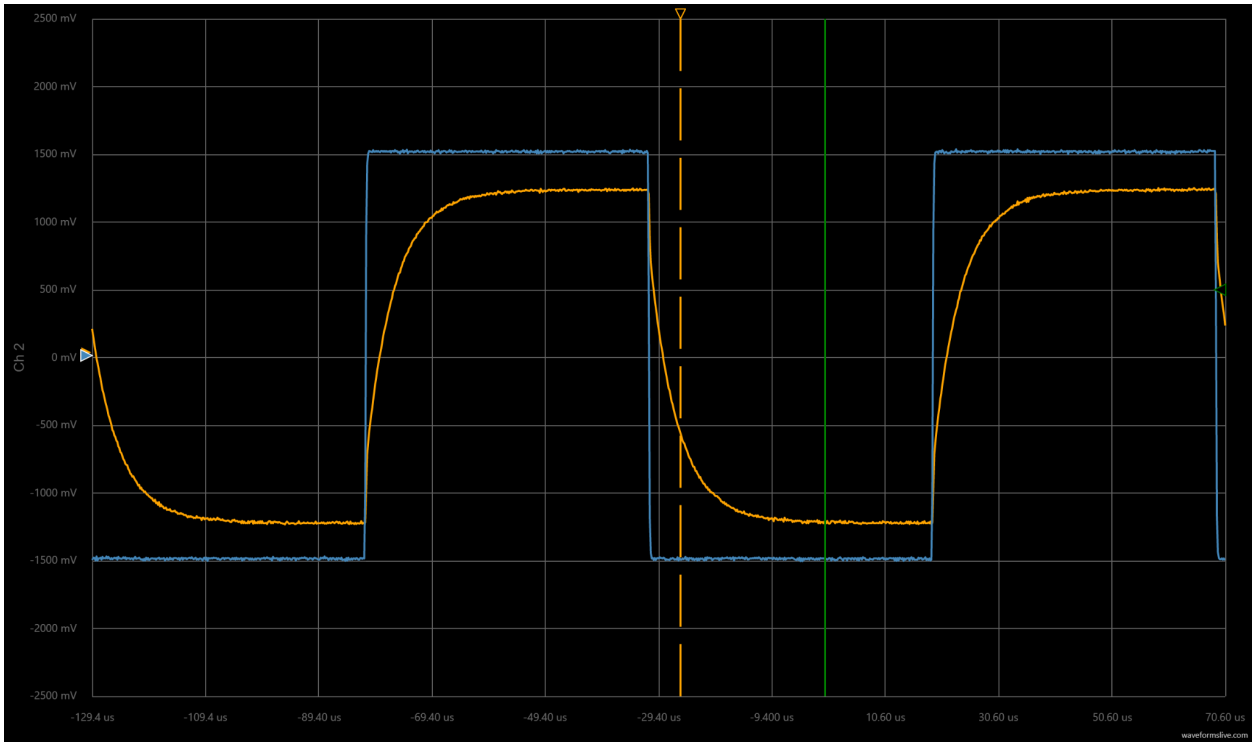
Capacitive Touch Sensor

The final sensor to learn to implement in this lab is the Capacitive Touch Sensor. The basic idea for capacitive touch sensors is that when you touch an object you effectively create an additional capacitance. The sensor used in this lab is a pad that has a gridded ground plane on the other side. A small capacitor is created when touched that induces a small capacitance change (in the pF range). We will be measuring this change in capacitance.

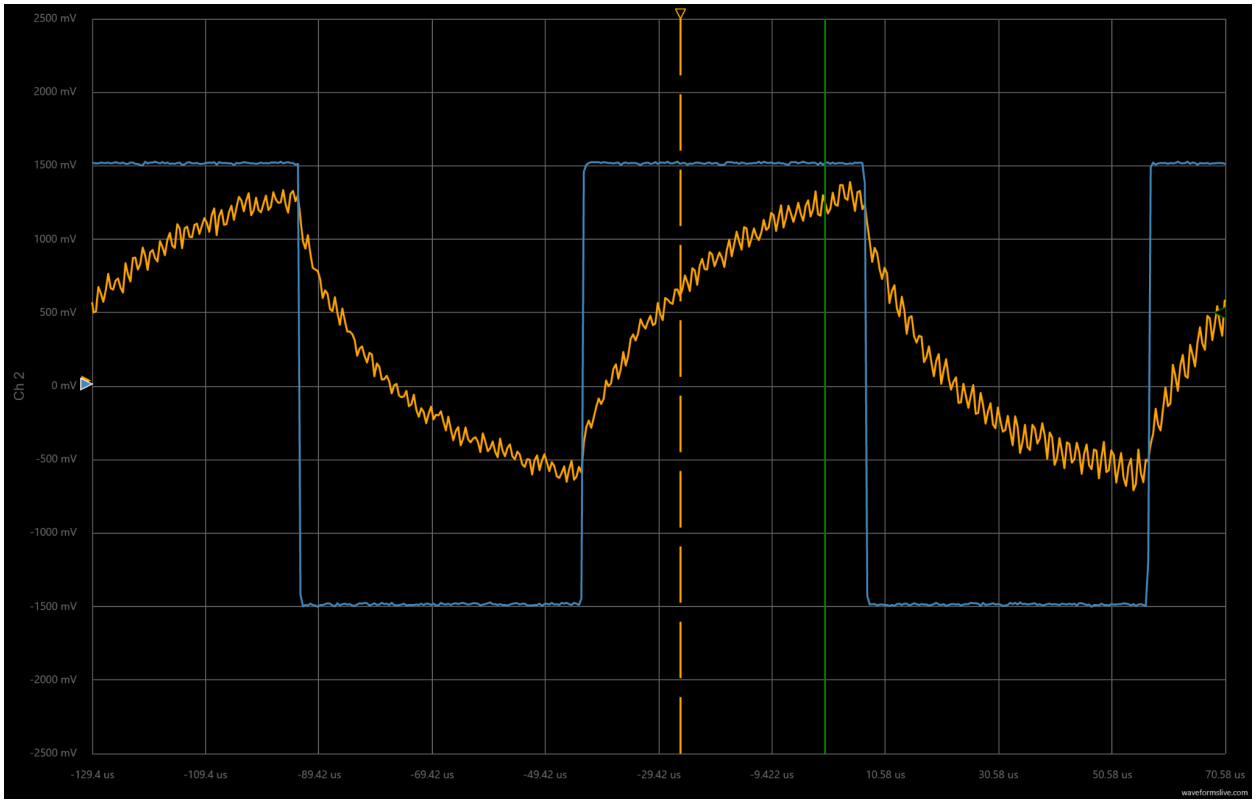
The first capacitance change we will measure is that of the RC time constant with the capacitive touch sensor in an RC low-pass filter configuration. We will place our capacitive touch sensor in a low-pass filter configuration with a 220 kOhm resistor. We can then observe the output with the OpenScope. The circuit diagram for this part can be seen below.



We can view the waveform and the changes that can occur when the capacitive touch sensor is touched. From this experiment we can gather the change in capacitance from touching the sensor. Below is the waveform we can see before touching the capacitor:



And then after touching the capacitor:

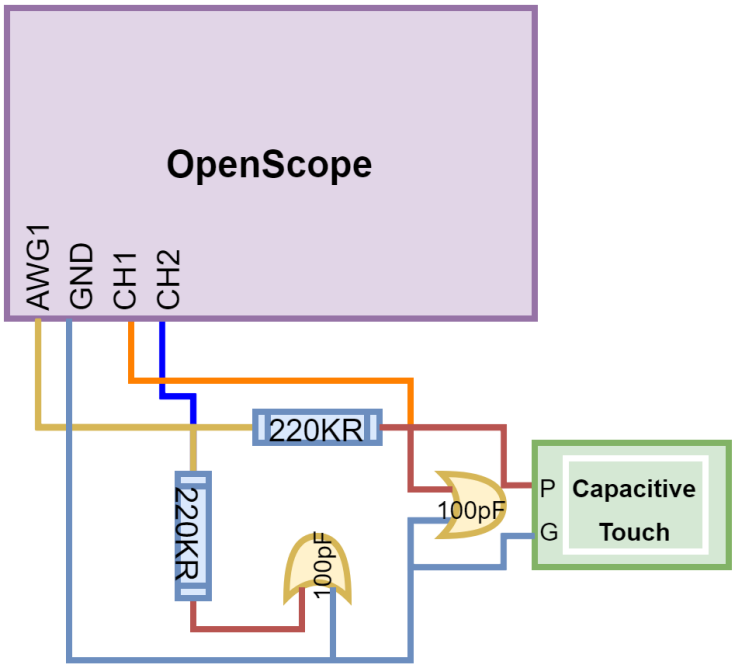


Using these graphs from the OpenScope we can calculate the capacitance of the touch sensor in both instances when it is touched and when it is not touched. Since we know that one time constant, RC, occurs at 63% of the height of our capacitance curve, we can use data gathered

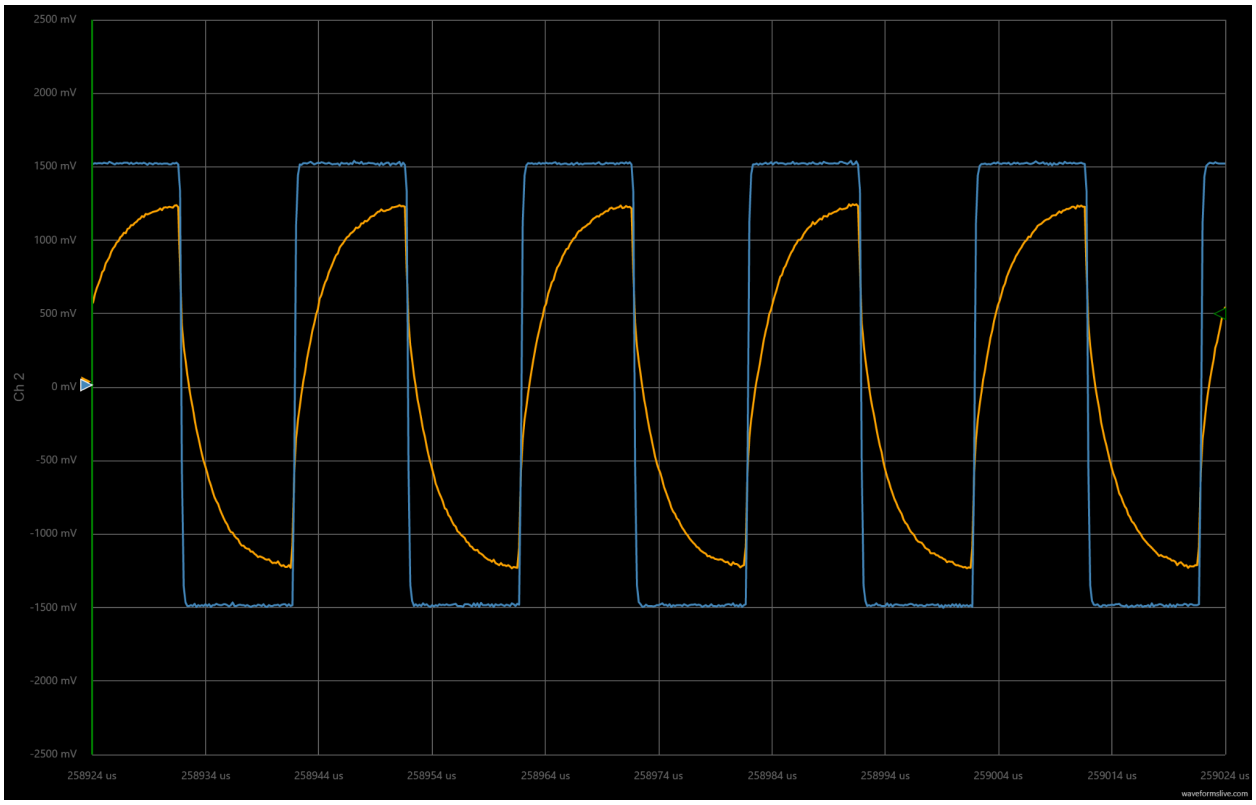
from our capacitance plots combined with our known resistance to solve for the capacitance. The peak-to-peak voltage of the capacitance signal is 1.43073V when the capacitive touch sensor is not touched. Using the cursors on WaveformsLive, we can place the cursors to give us the change in time from the bottom of the ramp up of the capacitance curve, to 63% of the ramp up of the capacitance curve. This gives us our time constant, 3.93 us. Using the RC time constant relationship (T = RC) and our known resistance (R = 220 kOhms) we can solve for capacitance (C). This gives us the capacitance of 17.864 pF for our capacitive touch sensor. For the capacitance waveform when the capacitive touch sensor is touched, we get a peak-to-peak voltage of 1.907V. Using the same method as previously, we find the capacitance (C) with the RC time constant relationship (T = RC). Using our cursors we find that the time it takes the output waveform to go from the minimum peak to 63% of the maximum voltage is 16.35 us. We can use the time constant RC relationship to find that the capacitance when the capacitive touch sensor is being touched increases to 74.318 pF.

The next configuration we want to test our capacitive touch sensor in is the Capacitive Bridge circuit. We will use two 220 kHz resistors and two 100 pF capacitors. We will be testing this circuit in a low-pass and high-pass configuration. In both configurations the capacitive touch sensor will be in parallel with one of the 100 pF capacitors. The circuit diagram for the low-pass bridge configuration is shown below. Hooking this circuit up to the OpenScope we can see the different waveforms generated with and without touching the capacitive touch sensor.

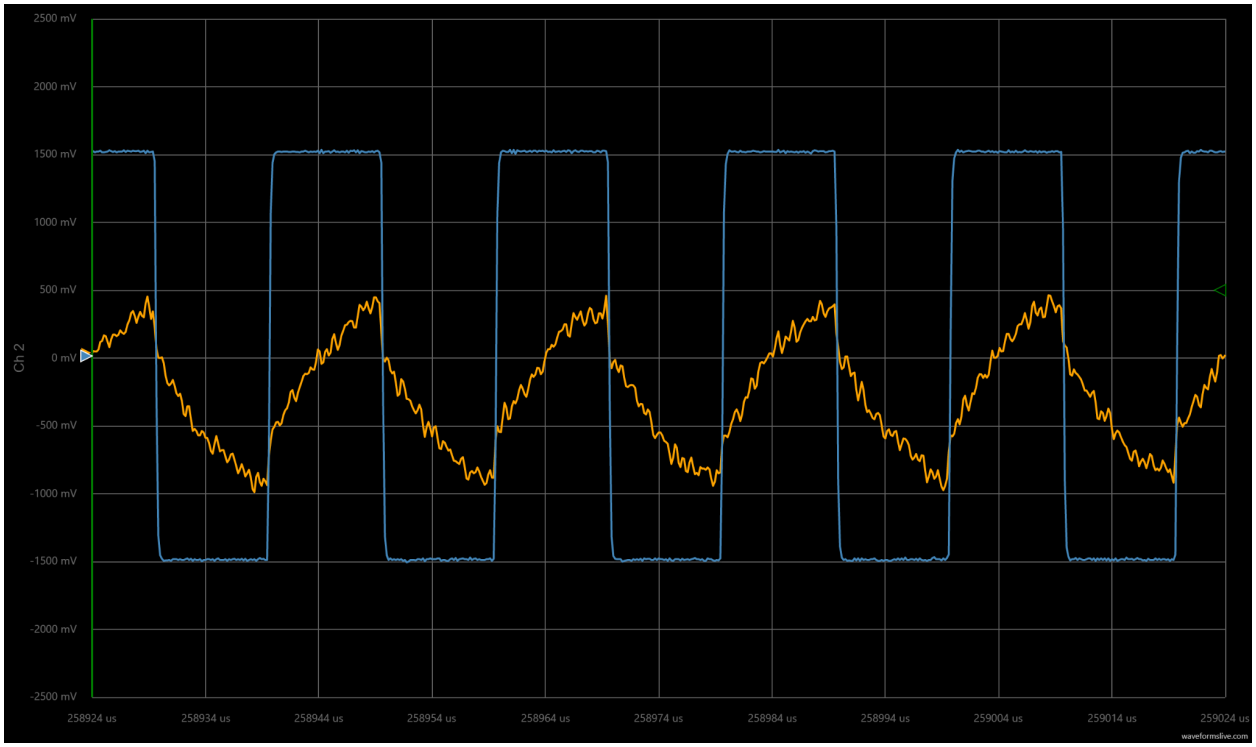
Capacitive Touch Sensor Low-Pass Bridge Circuit Diagram



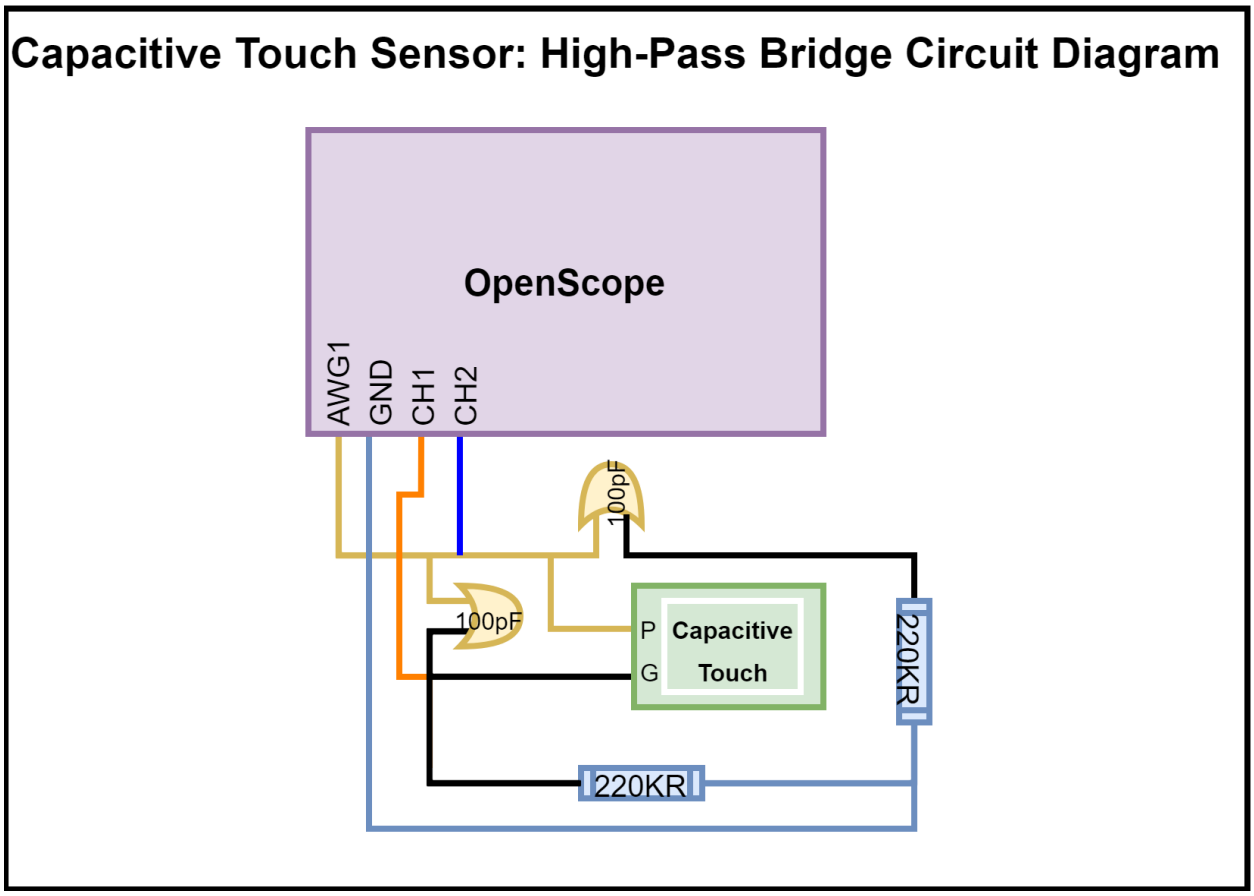
Below is the waveform of the output of the low-pass bridge configuration when the capacitive touch sensor is not touched.



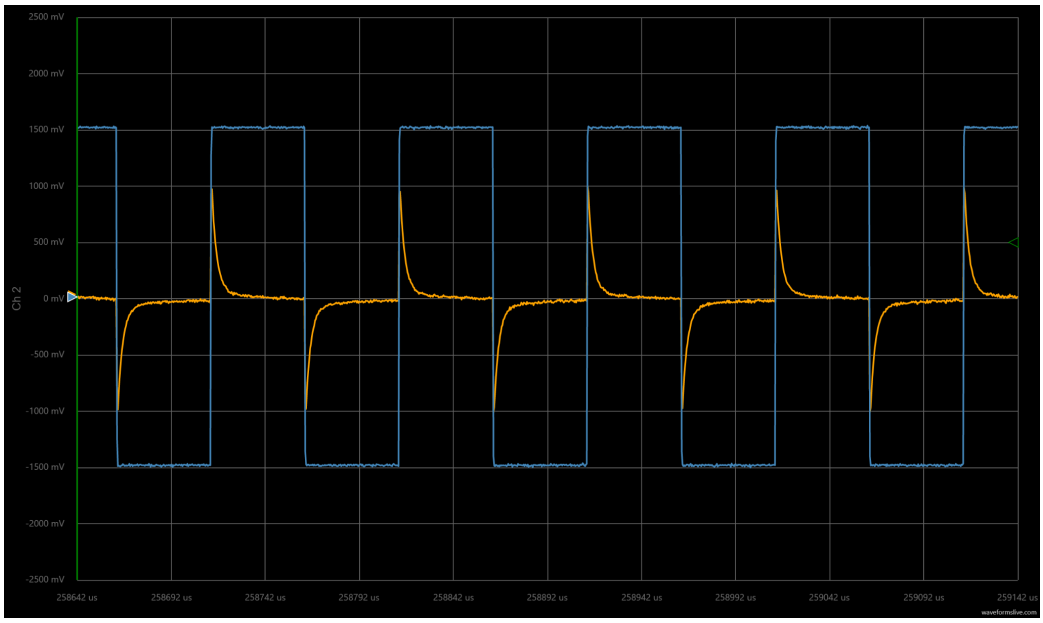
The output of the circuit when the capacitive touch sensor is touched can be seen below.



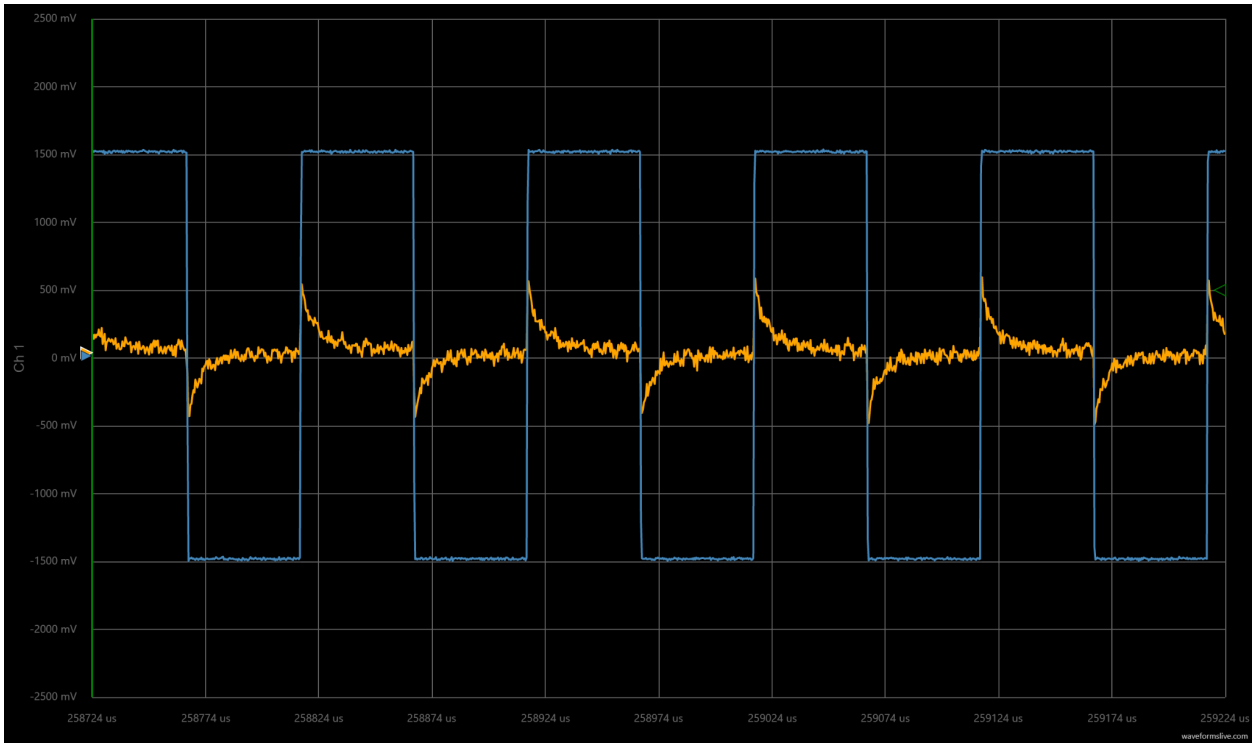
Then we can change the configuration of the circuit to be a high-pass bridge circuit and observe how that changes our waveforms. Below is the circuit diagram for the high-pass bridge configuration.



Below is the output of the high-pass bridge when the capacitive touch sensor is not being touched.

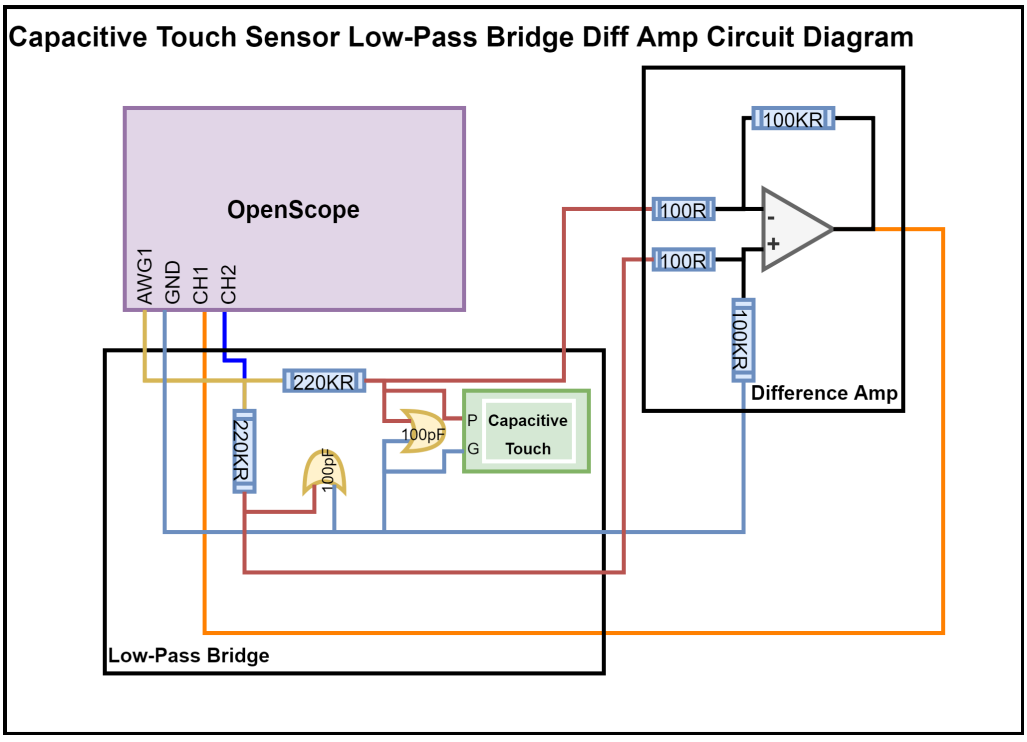


And then below is the waveform that is output when the capacitive touch sensor is touched.

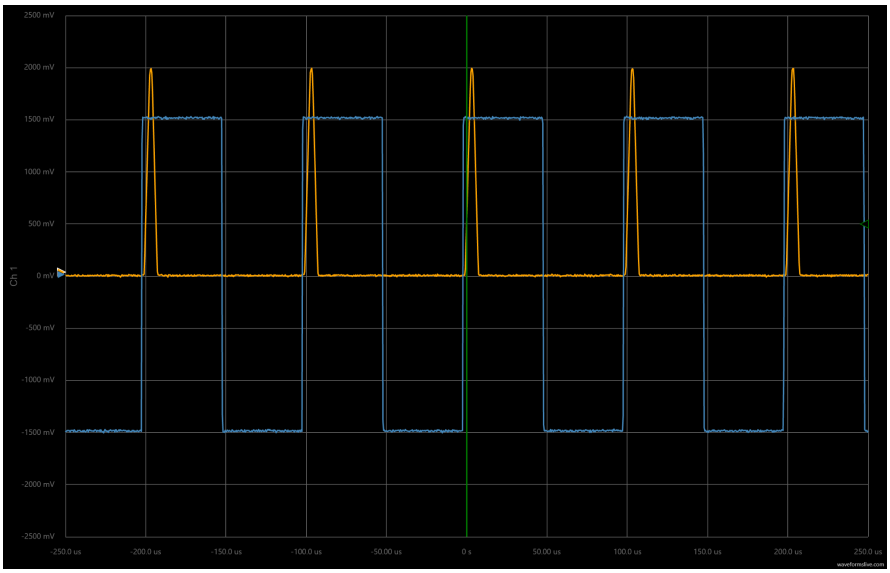


Comparing the two pairs of waveforms, we can see that the low-pass bridge configuration has a more visible change in the output waveform, so we will continue the lab with the low-pass version.

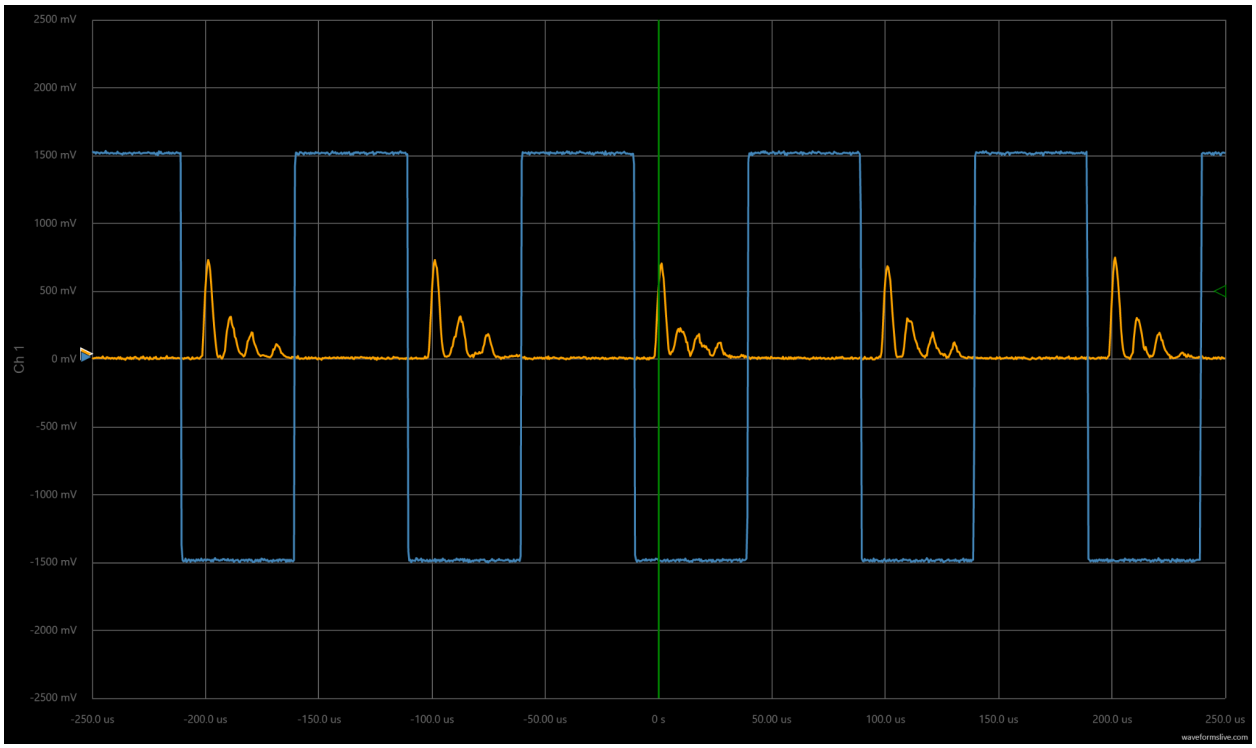
We can reconfigure this circuit to include a Difference Amp. The circuit diagram for the low-pass bridge with the difference amp can be seen below.



The output waveform of this circuit when the capacitive touch sensor is not touched can be seen below.

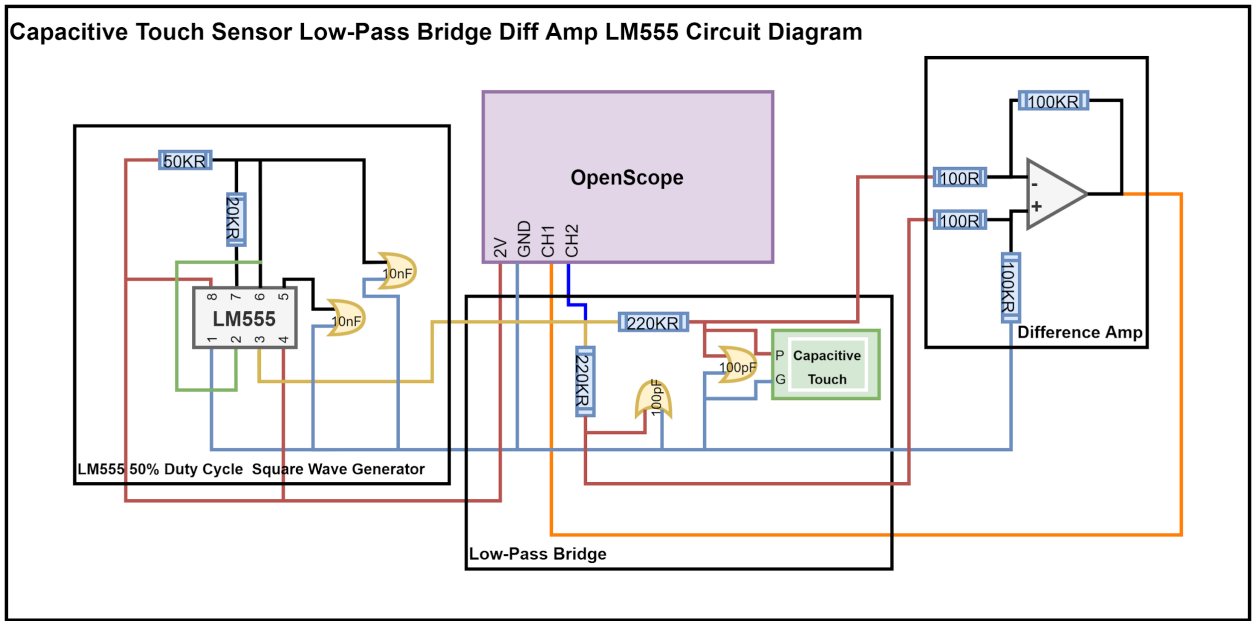


The output of the waveform when the capacitive touch sensor is touched is as follows.

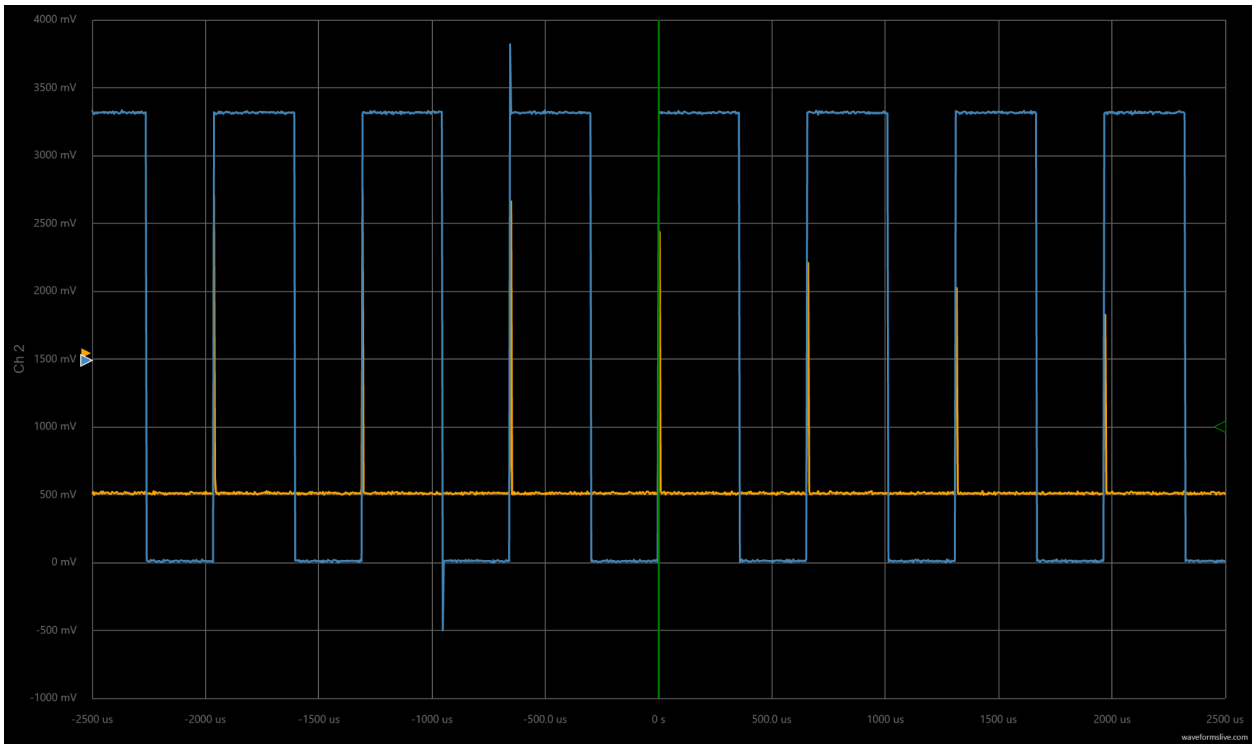


After this we implement the same circuit, but this time we use the LM555 to produce a 50% duty cycle square wave to drive our low-pass bridge configuration.The circuit diagram for this

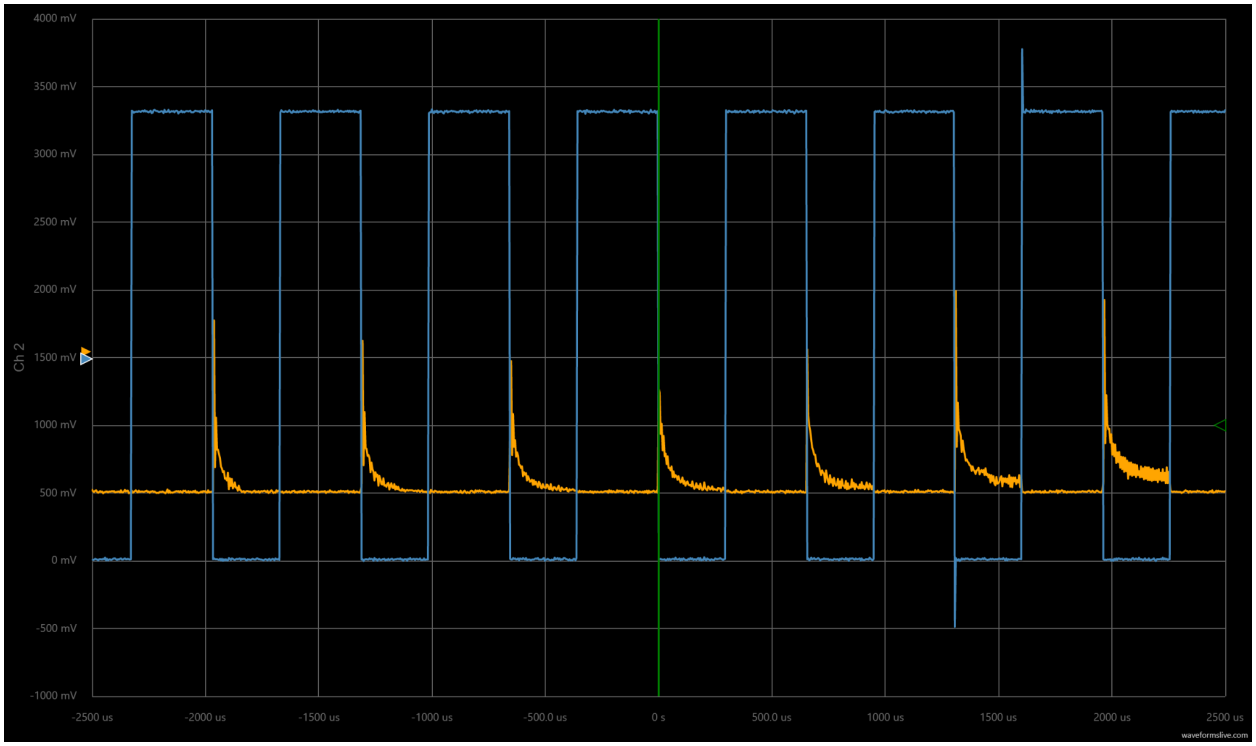
configuration can be seen below.



The output of this waveform without touching the capacitive touch sensor can be seen below.

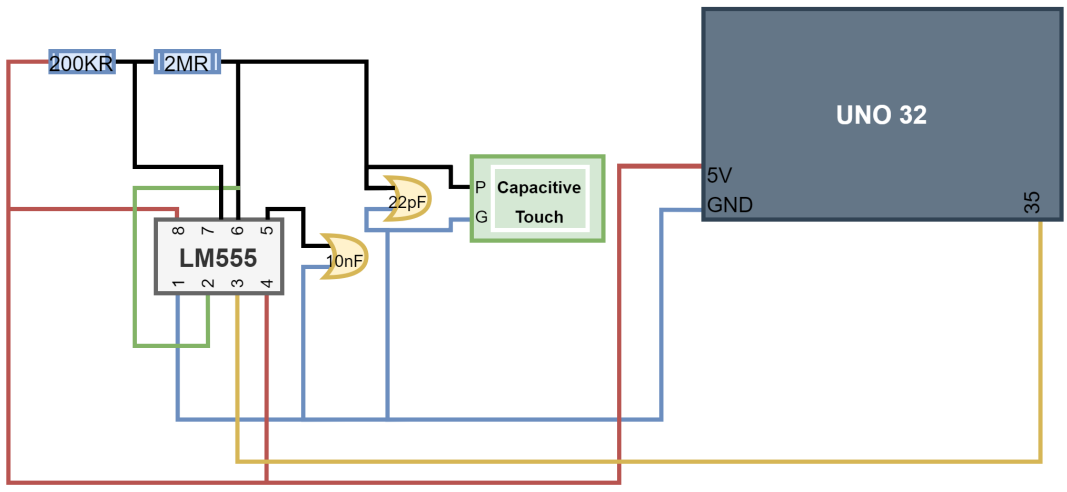


The output of this waveform when the capacitive touch sensor is touched can be seen below.

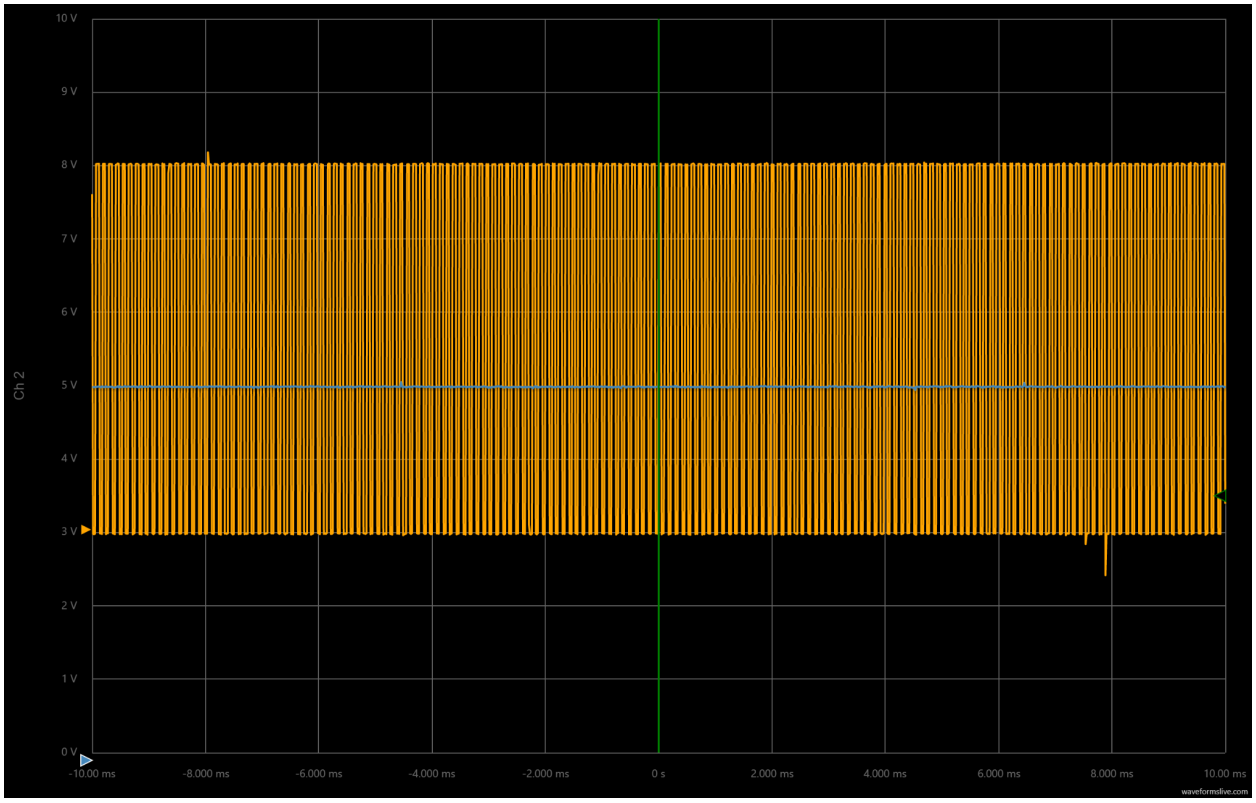


After testing this circuit out we can implement the Relaxation Oscillator. The circuit diagram for this configuration can be seen below.

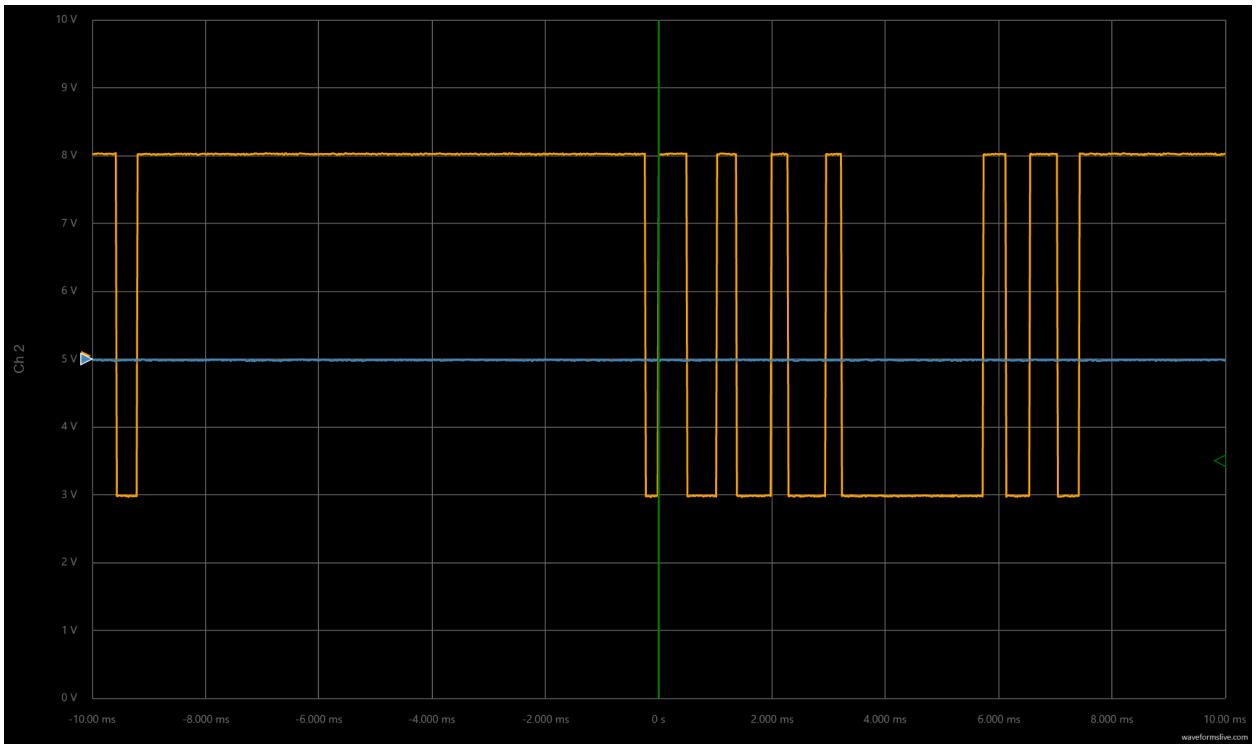
Capacitive Touch Sensor: Relaxation Oscillator



This circuit outputs a square wave. When the capacitive sensor is touched the frequency of the square wave decreases. We will be connecting the output of the LM555 into the UNO 32 to keep track of when the capacitive sensor is touched. Below is the waveform of the output when the capacitive sensor is not touched.



Below is the output of the LM555 when the capacitive sensor is being touched.



Now that the Relaxation Oscillator has been set up, we can complete the capacitive sensor software implementation. As in previous software implementations, we are given starter code. We will be using some module variables. We need a variable to set our trigger (t), a variable to

keep track of the timer count (c), a variable to keep track of the first count value (c1), a variable to keep track of the second count variable (c2), a variable to store the frequency (f), and a variable to store whether or not the capacitive touch sensor is being touched (touch).

The output of the Relaxation Oscillator is run into the Input Capture peripheral. On the IC interrupt we determine the period by subtracting the previous recorded time by the current time. If we keep track of this we can determine whether the frequency of the oscillator has changed, which indicates that the capacitive touch sensor is touched. In the CAPTOUCH.c file IC4BUF holds the current timer value. We use this to determine the time our square wave is on a rising or falling edge. We initialize our trigger state as zero within the CAPTOUCH_Init() function and we return SUCCESS. Within the CAPTOUCH_IsTouched() function we return our touch variable. Most of our work is done within the Input Capture ISR. Within this ISR we store the current count of the timer. Then we go into an if-else statement which depends on the value of the trigger variable (t). If the trigger value is one, we store the count of the timer (c) into c1 and set the trigger to zero, else the trigger value is zero so we store the count of the timer (c) into c2. Within the else statement we calculate frequency by subtracting c1 by c2 and store it into the frequency variable (f). Then we check to see if f is greater than our noise threshold. We have determined experimentally by printing out to the terminal that a frequency larger than 100,000 we do nothing. Else, if the frequency is greater than 1000, we know that the capacitive sensor is touched so we set the touch variable touch = TRUE. Else, if the frequency variable is less than 1000, we set touch = FALSE. At the end of this else statement we set the trigger flag to one so we can go into the next state on the next IC trigger. In our main we simply check if CAPTOUCH_IsTouched() returns TRUE. If it does it prints “Sensor is touched.” Otherwise, we print out “Sensor is not touched.” This concludes the CAPTOUCH software implementation.

Conclusion

In Conclusion this was a very long lab compared to the previous two. It was fun learning about these new sensors, but it was kind of annoying dealing with the noise of the PING sensor. It is apparent to me now that sensors will always have noise and that we should always be ready to perform some software filtering to get a better reading from our sensors.

The incremental nature of these parts were a good practice in incremental coding. It was useful to be able to learn about the effects of the capacitive touch sensor in different circuit configurations before implementing our software. It was important because it showed just how important the effects our circuit configuration can have on our sensor readings. These labs take a long time to understand. The information in the lab manuals takes some time to digest. I would advise anyone who has to do this lab in the future to read the lab manual as early as possible and let the information marinade for a bit as they attempt to get this lab done.