UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE167/L: INTRODUCTION TO MECHATRONICS

LAB 2: ENCODER, PING SENSOR, AND CAPACITIVE TOUCH

PURPOSE:

This lab is the introduction to quadrature encoders, time based sensors, and capacitive sensors. The encoder used in this lab is a rotary switch with RGB LED. The basic task for the students is to implement the quadrature encoder interface (QEI) to determine position and then drive the color as appropriate. The ping sensor is used to determine distance to various objects, and needs to be calibrated to generate accurate distances. Variable capacitance is used in myriad sensors, but in this lab we are going to detect the change in capacitance from touching a pad (so called capacitive touch sensors). These are used widely in user interfaces. In this lab, you will be implementing the QEI in software, driving the RGB LED, developing a Least Squares calibration for the ping sensor, and using various techniques to measure the change in capacitance, along with modular non-blocking code for each sensor.

There are three different types of sensors in this lab, and you are going to be exploring each one to understand how they work, and to create some modular code that will allow you to interact with them and read them.

As with previous labs, you will be implementing non-blocking code on the Uno32 to accomplish this. Different from previous labs, you will be implementing these sensors as modules (headers will be provided to you) to encapsulate the code and give you a complete module that can be used in the future to access these types of sensors in an efficient manner.

Note: this lab is going to be a large step increase in work over the previous two. Read the documents provided very carefully (and more than once) and plan out your approach for getting the lab done. You will save yourself many hours of lab time if you have a plan on what to do before you come into lab.

HARDWARE PROVIDED AND INTENDED USE:

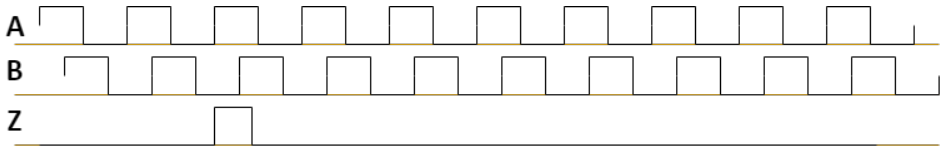
Uno 32, Speaker, Audio AMP (pot for volume), breadboard, **rotary encoder**, **ping sensor**, **capacitive touch board**, lm555 timer chip, mcp6004 quad opAmp chip, assorted caps and resistors.

BACKGROUND QEI:

Encoders (often called Incremental Encoders, or Quadrature Encoder [QE]) are used to measure angular displacement. Wikipedia (https://en.wikipedia.org/wiki/Incremental_encoder) has a decent reference on incremental encoders. Encoders are used in a wide variety of applications, from joint angles in robotics and keeping track of wheel rotation for navigation (odometry), to keeping track of axes on CNC machines and tracking tank turret angles.

They are called Quadrature Encoders because the signal consists of two square waves that are 90° out of phase with each other. By comparing the state of one of the square waves when the other is either rising or falling, you can generate a direction in addition to a count (or accumulated angle). Quadrature Encoder's are also called incremental encoders, because they give you no indication of absolute angle, rather

only accumulated angle from when you started measuring them. Some Quadrature Encoder's have an index mark that pulses once per revolution to give you a reference angle.[†]



The phases of the encoder are usually designated "A" and "B" and if there is an index mark, it is customarily called "Z." A typical trace of the outputs is shown above; it can be seen that if "B" is high when "A" is on a rising edge, the encoder is going in one direction, and if "B" is low when "A" is on a rising edge, then the encoder is going in the other direction. The beauty of the encoder is that there is no noise, and the signal does not drift. This makes it an excellent sensor for angles and rotation rates.

Ignoring the index pulse, "Z," the counting up and down can be formulated as a simple state machine from the values of "A" and "B" and triggered any time there is a change in either. Some microcontrollers possess a QEI subsystem that implements the state machine in hardware, but our variant of the Uno32 does not. It does, however, have both external interrupts and change notify pins, which will signal an interrupt any time the input state of the pin changes. Note that the simple state machine does not have a graceful error recovery. If the encoder skips a step, you really have no idea what happened and cannot easily recover[‡]. You will be implementing the QEI state machine inside the QEI.h/.c module, including using the change notify interrupts.[§] See the Appendix for more information on the Change Notify interrupt.

OPEN MPLAB-X AND CREATE A NEW PROJECT:

As with the previous labs, follow the CMPE167_MPLABXNewProjectInstructions.pdf instructions to create a new project for each section of the lab. Be sure you understand the difference between absolute and relative paths when making the project, or you will risk deleting your project when running the code updater.

PART 1 - READ QUADRATURE ENCODER INTERFACE (QEI) AND GENERATE ANGLE ACCUMULATION

The quadrature encoder will first need to be set up on your breadboard. To do this, see the Sparkfun documentation for the encoder.[**] You should be able to find a test circuit diagram. Consult with the TA if you have difficulty. You will also need solder your encoder to its corresponding PCB (again see the documentation on Sparkfun).

To read from the encoder you will have to read the two output signals, A and B. They are offset from one another by 90°of phase (hence: quadrature). By reading which signal is high and low with a state machine (or similar logic) you can determine the direction of the encoder and its position (how much it has turned since you last reset it). You will have to read both positive and negative increments of the encoder. The count should be incremented for clockwise rotations, and decremented for counter-clockwise rotations.

[†] If you need to know absolute angle, then there is another kind of encoder known as the absolute encoder which measures the angle directly. They are usually limited in resolution due to their construction, and use a "gray" encoding method that ensures that only one edge transitions on any rotation. This is so the position does not jump due to timing differences in the sensor parts.

[‡] If you do get a missed step (as in the sequence in your states doesn't work), simply reset the state and keep counting. You will lose that step, but otherwise it is fine.

[§] The QEI.h header file contains relevant implementation details in the comments and should be studied carefully. As always, you want to ensure that your code is NOT blocking, and that reading the QEI sensor gives the last known count on the encoder.

[**] https://cdn.sparkfun.com/datasheets/Components/Switches/EC12PLRGBSDVBF-D-25K-24-24C-6108-6HSPEC.pdf

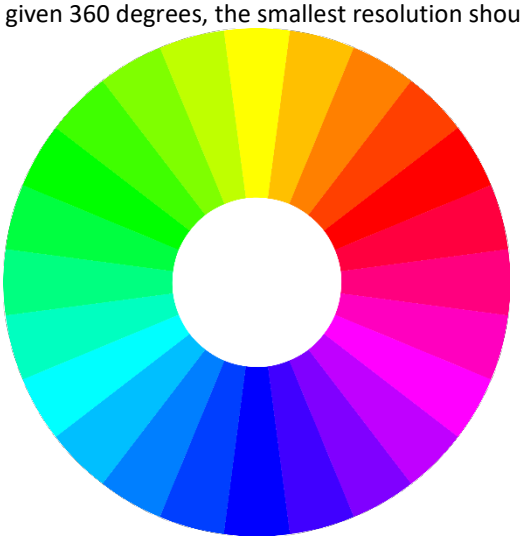
Using the count, you will need to map the degrees of the rotation from 0 to 360 for one rotation clockwise, and 0 to -360 for one rotation counter-clockwise.

It is up to you to roll over the encoder at ±360° or to continue to count up or down. This specific encoder produces 24 pulses per revolution on each channel (A and B), and thus can generate a resolution of 4x that (or 96 counts per revolution). This is done by triggering your QEI state machine on every transition of the encoder signals.

QEI Software Implementation: you will be using the change notify (CN) interrupt to generate your count. Each time the input is generated, you need to read the port, update your count as appropriate using your state machine, update the state, and exit the interrupt. The count should be held in a module variable. A QEI_ResetPosition() should reset the count (module variable), the state of the QEI state machine, and the CN interrupt to trigger on the next change such that it now counts from zero.

PART 2 - MAP QEI TO COLOR

Once you have successfully gotten your encoder to count up and down as you turn it clockwise and counter-clockwise, you are going to change the output of the RGB LED based on the rotation. There is quite a bit of science on colors and how to represent them, if you are curious see the Wikipedia article on color (https://en.wikipedia.org/wiki/RGB_color_model). Using your readings from the encoder, you will map the output angle to a color wheel, as shown below. The encoder has RGB LEDs within the knob. You will use PWM signals to change the color of each LED proportionally, so that as you turn the knob of the encoder, you can seamlessly simulate the color wheel. For example, if the knob has not been turned, the initial color would be yellow; turning the knob about 45 degrees clockwise should begin to change the color to orange; another 45 degrees should make the color red. Find out what the highest resolution for the encoder is. Make sure that the smallest degree of rotation changes the color by some amount. The encoder has 24 pulses per rotation, so given 360 degrees, the smallest resolution should be easy to calculate.[††]



Note that how you wish to fade from one color to the next as you turn is entirely up to you. There are quite a few variations on this (and quite a bit of science behind it). Realize that some of them are quite simple. The suggestion is to keep it simple at first, and go back and embellish later if you want to geek out about it.

[††] By triggering on every edge of the "A" or "B" signal, you should be able to quadruple the count on the encoder per revolution. That is, once direction is determined, the count is incremented (decremented) by XOR'ing A and B. Though you will find that you would only do that if implementing the QEI in hardware.

BACKGROUND PING SENSOR:

The second sensor used in this lab is a time-of-flight ranging sensor called a "ping" sensor. Time of flight sensors work by injecting some form of energy into the environment and measuring the time of flight between the emitter and the reflection (echo) from the object. The kind of energy and the method of measuring the time of flight differentiate the various sensors, ranges, costs, and accuracies. These sensors are used widely in a variety of applications, from mm ranges (VCELS) to 100s of kilometers (RADAR). Typically, light and sound (at various frequencies) are used as the source. For ranging, techniques range from simple timers, signal strength, frequency matching, and interferometric techniques. A decent background document available at: https://en.wikibooks.org/wiki/Robotics/Sensors/Ranging_Sensors shows many of these techniques and how they work.

Fundamentally, all time-of-flight sensors measure distance by multiplying the time between sending the pulse and measuring the return by the speed of flight in the medium. That is: $d = 0.5 \times v \times \tau$ (the 0.5 is to account for the fact that the echo covers the distance there and back. The specific range sensor in this lab uses ultrasound (~40KHz) way above human hearing, and uses a chirp function (both frequency and amplitude modulated) to get better discrimination on the return timing. They are low cost, and generally work quite well.

Looking at the timing diagram in the datasheet,[‡‡] you can see there are several things happening that lend themselves to a state machine triggered by timing and the change notify subsystem of the Uno32. The first thing to note is that the trigger input needs to be raised for at least 10µs and then lowered again. Following this, the module will transmit a burst of ultrasonic pulses, and will generate an output pulse whose high time is proportional to the distance (they specify the time of flight as 340m/s). The datasheet also specifies the maximum range of 400cm (corresponding to 11765µs), and a delay of at least 60ms between triggers (or a max repetition rate of 16.667Hz). Note as well that the output pulse may not come at all if the device cannot generate a valid echo timing.

Thus, your state machine must (roughly): raise the trigger line and lower it 10µs later, reset the timer to interrupt 60ms later to start the ping cycle again. On the change notify interrupt, read the µS timer, and use this to calculate time of flight. Your user code will convert time of flight to distance.[§§] You will be implementing this in non-blocking, interrupt driven code within the PING.h/.c module.[***] See the Appendix for more information on timing interrupts and the Change Notify interrupt system.

PART 3 - PING THE PING SENSOR AND CAPTURE OUTPUT

Set up the software to control the ping sensor to measure distance. The basic idea behind doing this is to trigger the ping sensor every 60msec using the timer. Using the Change Notify interrupt and the µS timer, infer the distance based on the elapsed time. Your code should be able to output distances to the terminal so you can read them. Take measurements with known distances (use a ruler or similar), and record the time it takes to sense a ping to a wall, for instance. If you can back-calculate the distance, based on elapsed time, and it matches the actual distance to the wall, you can trust your ping sensor. Experiment with different objects and see if the sensor loses the signal at certain angles and distances. Get familiar with the sensor. Demonstrate your ping sensor working to the TAs.

Ping Sensor Software Implementation: you will trigger the ping and use the timer to generate an interrupt ~10µs later to drop the trigger ping (ensuring that you have held the trigger up for at least 10µs). During

---

[‡‡] https://cdn.sparkfun.com/assets/b/3/0/b/a/DGCH-RED_datasheet.pdf

[§§] Any of you who decide to do a floating point multiply by 0.5 or floating point divide by 2.0 will immediately fail the class. And we will go back and retroactively fail you for CE13 as well. Remain in integer math as absolutely long as possible and don't use floats for simple calculations.

[***] Again, read the comments of the header file carefully as many of the specifications are there.

the interrupt, you will reconfigure the timer (using the PRx register) to generate another interrupt ~60ms later to ensure that you don't hit the trigger too often (see the datasheet for specifications). During the long delay, when the CN interrupt triggers, you will be reading a free running µs ticking counter and using that to calculate the time of flight.[†††] On the timer interrupt, you will raise the trigger pin, reset the PRx register to its appropriate value, and repeat. Note that this looks a whole lot like a simple two-state state machine.

It has been observed that there are glitches on the echo line during the trigger pulse. It is important that in software you make sure that your reading is not being accidentally glitched by the trigger pulse. Also you can use outlier detection if you have readings that are way out of bounds.

PART 4 – CALIBRATE THE PING SENSOR USING LEAST SQUARES

The datasheet gives you a basic equation for distance using the specified 340m/s time of flight for the sound waves in air (at sea level and standard temperature). This is a linear relationship; however, you are going to calibrate your sensor to give you a direct equation from measurement to distance. This is both to give you a handle on Least Squares (very powerful technique for calibration) and to do so on a relatively benign sensor; plus, you get a more accurate sensor. Note that other optical distance sensors have a non-linear relationship to distance which would require a different measurement model and parameter set.

A quick primer on Least Squares can be found on Wikipedia (https://en.wikipedia.org/wiki/Least_squares) and is worth the read. Least Squares was developed by Legendre and Gauss (before they had access to linear algebra) and underpins an enormous amount of engineering in the world. It is particularly useful when you have many more measurements than you have parameters, and finds the best approximation to the underlying function that minimizes the squares of the error between your model and the measurements (hence the name, least squares). The model is not restricted, as long as it can be parameterized by some set of (unknown) parameters. In this lab, we will develop least squares to estimate the distance to the target.

In the case of fitting a line, there are two parameters, the slope (*m*) and the intercept (*b*). With two parameters, you might think that two measurements would be ideal; from there you could calculate the slope and intercept exactly with no errors. The problem comes in with noise. Your parameters are sensitive to noise, and this effect would be invisible if you had only two measurements. Instead, we will use multiple measurements to minimize the effect of noise.

Least squares begins with a set of data pairs (x_i, y_i) where x_i is the known distance (using a ruler, tape measure, jig, etc.) and y_i is the output of the ping sensor. In the lab, you will generate a number of data pairs using your software developed above. You want to make sure that you cover the complete range of distances you are interested in so that you have good coverage. Also note that the units of y_i don't matter (i.e.: y_i could be in meters, µs, or counts) in that the conversion to distance will be taken care of by the slope.

The equation of the line is: $y_i = mx_i + b$ where *m* is the slope and *b* in the *y*-intercept. If *m* and *b* were known, then for any given output from the sensor, $\hat{x} = (y_m - b)/m$. Thus the best estimate of the distance ($\hat{x}$) is using the best estimate of those parameters $\hat{m}$ and $\hat{b}$. Least squares allows us to estimate those parameters from our set of data pairs. This begins by rewriting the equation of a line in matrix form:

[†††] Here you will be reading the value of a fast free running timer that is counting microseconds, thus you will need to keep track of the current and previous values to get an elapsed time.

$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_\vdots \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_\vdots & 1 \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} m \\ b \end{bmatrix}}_{p}$ where the data pairs are inserted into the matrices Y and X for all measurement pairs.

The solution to the matrix form of the equation above that gives the least squares *estimate* for *m* and *b* is: $p = \begin{bmatrix} \hat{m} \\ \hat{b} \end{bmatrix} = [X^T X]^{-1} X^T Y$. These are called the "normal equations" and their derivation is beyond this lab. Nor does Matlab actually implement them in this way, but does something more numerically stable.[‡‡‡]

Finally, for this lab, create for yourself a target (foamcore works well) and mark distances in the lab that are repeatable using a tape measure or other ruler. Set your target at a known distance and have your software output values of the sensor measurement.[§§§] Capture this data into a file so that you can create the X and Y matrices in Matlab. Generate the best estimates for your parameters, and then code a function to convert your measurements to distance. Redo your experiment (but now with the least squares conversion) and see if the errors you see at each known distance can be quantified. Are they uniform, do they change with distance, are they white? Play with the sensor and get a sense of how it works.

Include a plot of the raw data and the fitted line in your lab report, plus a description of the setup and any interesting observations you had.

PART 5 - TONE OUT BASED ON DISTANCE:

Now that you are able to calculate distance with the ping sensor reliably and accurately, write a program to change the tone of your speaker (same speaker setup as the previous labs) based on distance to the ping sensor. You should consider having set distance (or range) that you can move your hand back and forth within to change the speaker tone easily. Keep the distances relatively short (< 3-4ft) so that you can have something like a Theremin.[****] Demonstrate this to the TAs, include the code in your lab report.

BACKGROUND CAPACITIVE TOUCH SENSOR:

The last section of the lab will be interfacing to a capacitive touch sensor. Many sensors rely on variable capacitance to sense the output, and learning how to measure capacitance is an important part of any sensors discussion. Capacitive touch sensors are used in myriad applications because of their many advantages: the switch is simple to build, has no moving parts to wear out, generates no spark (important for explosive environments), and can be placed behind a protective barrier such as glass or rubber such that the electronics are sealed from the environment.

The basic idea for capacitive touch sensors is that when you touch an object (or place your finger in close proximity) you effectively create an additional capacitance. See the Wikipedia article for a basic explanation of the phenomenon: https://en.wikipedia.org/wiki/Capacitive_sensing. We are using projected capacitance by using a pad that has a gridded ground plane on the other side; a small capacitor is created that when touched creates a small capacitance change (pF and unstable). This change in capacitance is what you will be measuring, and there are various ways to do this measurement.

Straight RC Time Constant: The first (and perhaps simplest) measurement method to understand is a straight measurement the RC time constant. That is, create a low pass filter with a known resistor, and drive it with a step change (or square wave). Compare the output of the low pass filter to some reference

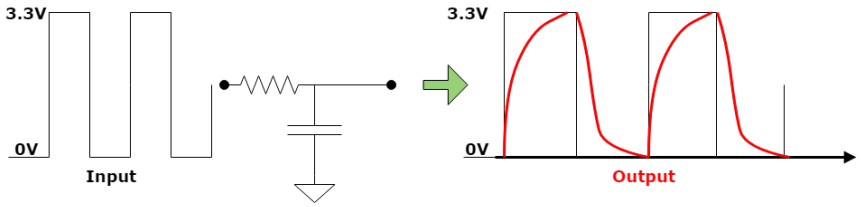
---

‡‡‡ In Matlab, you would use: *p = X\Y*; and extract $\hat{m}$ and $\hat{b}$ from the vector *p*. If you really want to have fun, you can go through the normal equations analytically and see what are the terms actually being calculated. The math is quite interesting and instructive.

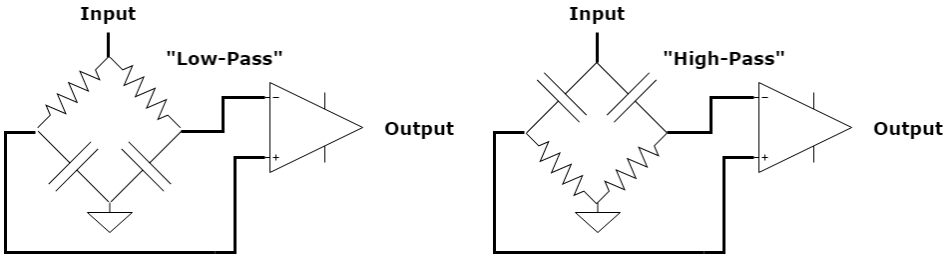
§§§ It is perfectly acceptable (indeed preferable) to have multiple measurements at each known distance.

**** https://en.wikipedia.org/wiki/Theremin

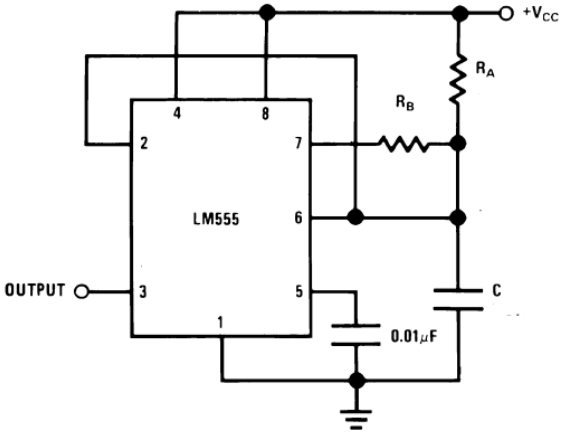
voltage, and you can back out the capacitance from the rise time. This depends on knowing the value of the resistor, generating the input square wave, and accurately measuring the rise time of the output. This can be a challenge when using very small capacitance as the time constants are very short unless you put in huge resistors, but that has its own problems. You must match the drive frequency to the RC time constant.



Capacitive Bridge: By setting two low-pass or high-pass with a known and the variable capacitor in a bridge configuration, and then sending the outputs through a difference amp, you can get rid of many of the noise sources and create a more stable measurement. This still requires driving the bridge with a square wave, and has the additional hardware in the form of a differential amplifier, but has many advantages over the RC time constant measurement. In the specific case of the capacitive touch sensor used in this lab, you will be hooking the touch sensor in parallel with a normal capacitor on one side of the bridge.



Relaxation Oscillator: Both the bridge and the RC time constant measurements are driven by a fixed square wave. The relaxation oscillator essentially runs the output of the low-pass through two comparators into an S-R latch (flip-flop) that is used to drive the input square wave. This is also known as a multivibrator, and implements a non-linear feedback that keeps the circuit oscillating. The advantage of this is that the feedback is dependent on both the charge and discharge of the capacitor, and creates a variable frequency square wave depending on the capacitance. See the Appendix on LM555 for more information.



In this lab, you will be using all three methods for determining if there is a touch on the capacitive sensor or not. You will be evaluating the different methods, and writing code to complete the CAPTOUCH.h/.c module using timers and input capture to implement the relaxation oscillator in a non-blocking, interrupt driven mode. For more information on these interrupts, see Appendix B.

PART 6 – RC TIME CONSTANT BASED MEASUREMENT

You are first going to measure the capacitance change in the touch sensor using a straight forward RC time constant. First, place your capacitive touch sensor and a large (greater than 100K) resistor as a low-pass filter. Drive this with a signal generator and watch the output change as you touch the pad on the sensor. Experiment with the frequency and amplitude of the drive wave, the polarity of the touch sensor, which side you touch, and the size of the resistor. Again, the point here is to play with the sensor and get some intuition about how the detection circuit works. Include images of the oscilloscope and commentary in your lab report. Measure the change in rise time and calculate the change in capacitance. Show the changing RC to the TAs.

PART 7 – CAPACITIVE BRIDGE AND DIFFERENCE AMP

You are going to implement the capacitive bridge and difference amp to measure the change in capacitance. There are several options that can be used to generate the best signal, and in this section of the lab you are going to experiment with all of them to discover the best to use.

Create the first bridge with the resistors on top and the capacitors on the bottom (use approximately 100pF caps). Place the touch sensor in parallel with one of the capacitors (as shown in the circuit diagram) and drive the top of the resistors with the signal generator. You may use the math capability of the oscilloscope to create the output of the difference between the two sides of the bridge if available; or you can simply put both sides on the oscilloscope and eyeball the difference. Try swapping the polarity of the touch sensor. Try touching it on either side. Vary the frequency of the drive. Again, the point here is to explore the sensor and see what it is doing.

Next, flip the bridge from two parallel low-pass filters to two parallel high-pass filters (that is, the capacitors now go on top and the resistors on the bottom). Now drive it from the capacitors using the signal generator and again experiment with polarity, frequency, and the two fixed capacitor values to see how the circuit behaves. In your lab report describe what you saw, include traces from the oscilloscope, and which configuration (and polarity) gave the best signal on the oscilloscope.

Now that you have determined the best bridge configuration, implement a difference amp using the MCP6004 with appropriate gains to get a useable signal.[††††] Verify with the oscilloscope and the signal generator that you are getting the output you expected. Now that you have determined the best configuration for the drive and sense circuit, implement the drive using an LM555 in 50% duty cycle mode (see LM555 appendix). Again, show both input and output on the oscilloscope. Show this to the TAs, and include the oscilloscope traces in your report.

Now run the output of the difference amp into an ADC pin on the Uno32 and write software to determine if there is a touch or not. Given the noise on the signal, it is likely you are going to need some simple digital filtering techniques to amplify your signal to noise. It is OK to use blocking code in this section. In your lab report, include details of how you implemented the capacitive bridge and your results.

[††††] We are actually agnostic on your using the MCP6004; you can go ahead and buy yourself an actual difference amp and use that, but we will only supply you with the 6004. Note that you will learn more building it yourself.

PART 8 – RELAXATION OSCILLATOR

For the last part of the lab, you are going to implement the relaxation oscillator using an LM555 chip, with the timing capacitor being the parallel of a fixed 22pF capacitor and the touch sensor. The LM555 in astable mode exactly implements the relaxation oscillator, with the thresholds set at 2/3 and 1/3 V_{cc}. See the LM555 Appendix for more details on how to set up the astable mode.

The timing capacitor between the threshold pin (6) and ground (1) is going to be a parallel 22pF capacitor and the touch sensor. Make sure you use resistors such that the base capacitance oscillates at a frequency that is reasonable (within the 1-20KHz range). You might need to experiment here. Use the oscilloscope to verify the frequency, duty cycle, and shift in frequency when you touch the sensor. Again, play with the circuit and see what the effects are. Switch the polarity of the touch sensor and see if it makes a difference.

Once you have the LM555 relaxation oscillator changing frequency when you touch the sensor, run the output into the Uno32. You are now ready to implement the CAPTOUCH.h/.c module that will use the input capture interrupt to measure the period of the incoming square wave.[‡‡‡‡] Think carefully about what you need to do to detect the presence or absence of a finger on the pad. Again, for more detail on the Input Capture and Timers, see Appendix B.

Capacitive Touch Software Interface: For the capacitive touch, you are going to use input capture to measure frequency (period). Recall that the relaxation oscillator creates a variable frequency (but constant duty cycle) square wave into the micro.[§§§§] The method used here will be averaging to increase the signal to noise. The output of the relaxation oscillator is run into the Input Capture peripheral. On IC interrupt, determine the period (current time – last time), average that reading with N previous ones, and dump that into a module-level variable. Note that you can also keep the N-past readings in the module variable, entirely up to you. When checking CAPTOUCH_IsTouched () you can do the average and compare it to a threshold.

You will find for the capacitive touch sensor that the signal is extremely noisy and will require filtering (averaging) to get a useable "touch/no touch" indicator. There is a tradeoff between the number of samples to average (more latency to touch) vs signal noise. Play with this to get a good sense of where the sweet spot is for your setup. Again, keep your frequencies on the relaxation oscillator lower than about 20KHz (you can adjust this with your resistors on the 555).

In your lab report, explain (in detail) what you did, how you did it, and how well it worked. Include plots from the oscilloscope, circuit diagrams, explanations, etc.

CHECK-OFFS

1. Demonstrate encoder can count up and down, show that QEI.h/.c works as required. Encoder should not lose counts when being moved (see Part 1)
2. Demonstrate the color change of RGB LED based on encoder counts (see Part 2).
3. Demonstrate Ping Sensor (accurate to better than 1cm); show that Ping.h/.c works as required (see Part 3 and Part 4).
4. Demonstrate Tone from Distance (see Part 5).
5. Demonstrate Capacitive RC change on oscilloscope (see Part 6).
6. Demonstrate Capacitive Bridge change on oscilloscope and in software (see Part 7).
7. Demonstrate LM555 Relaxation Oscillator on oscilloscope and in software. Capacitive.h/.c works as required (see Part 8).

---

‡‡‡‡ Again, read the comments of the header file, CAPTOUCH.h, carefully as many of the specifications are there.

§§§§ While you could low-pass filter the signal and send it into the ADC, you would have to match the cutoff of the filter pretty carefully and would have a very poor signal.

8. There should be three separate projects demonstrating the QEI, Ping, and Capacitive modules. Each should be well documented and commented.

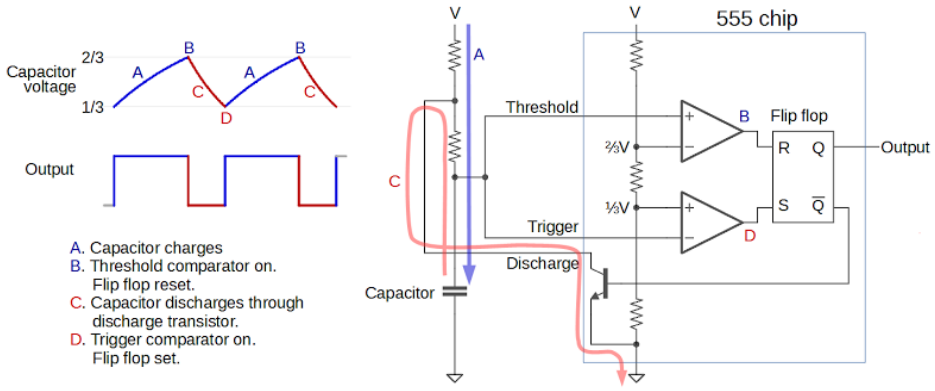
PART 9 - YOUR LAB REPORT:

Congratulations! You have now completed Lab 2 (the third lab). Make sure to explain each part of this lab in your report in enough detail that someone who hasn't taken this class could easily reproduce your results. We expect this to be written in full sentences, with decent grammar, and correct spelling.

APPENDIX A: LM555 OPERATION AND ASTABLE MODE

The LM555 is an old and very useful chip. Entire books have been written on all the various circuits you can make using the 555. At its core, there are two comparators, an S-R latch (flip flop), and some drive stages. While it can do a great many things, the most common use of the 555 is to implement an oscillator with a fixed period and duty cycle. This is known as astable mode (or a relaxation oscillator).

In essence (though not in actual implementation), a low-pass (resistor + capacitor) is driven by the output of the S-R latch. V_{out} is set at the input to two separate comparators (one set at 2/3 and the other at 1/3 of V_{cc}), and the output of the top and bottom comparator drive the R and S inputs of the S-R latch respectively. The latch starts out "high" and drives the RC circuit up (with the characteristic RC curve). Once the voltage exceeds the top comparator threshold, R is asserted and the output of the latch flips to "low," discharging the RC circuit. Once the input voltage drops below the bottom comparator, the latch flips to "high," and the cycle restarts.



The LM555 implements this relaxation oscillator circuit internally and is how we will be doing this in the lab (and in every other instance where you need one). The exact implementation is slightly more complicated, but works essentially the same (with some buffering to ensure that the output is protected from the RC circuit).

The data sheet goes through the math that is used to determine the rise time, fall time, period, and frequency as a function of the two resistors (R_A and R_B) and the capacitor (C). Because the charge and discharge paths have two different resistors in the way, it is quite challenging to get to a 50% duty cycle. The right ratio of resistors can get close to 50% duty cycle, but if you are going for 50% duty cycle, you need to use a variation of the astable configuration. It isn't really necessary for what you are doing in this lab, but it is a good circuit to build anyway.

The 50% duty cycle circuit is found only on the old national datasheet (happily we have a copy of it on the class CANVAS site), and contains the remaining equations that you need to match to get the right ratio of resistors. Going through standard components can take a while to get it all to match.*****

***** **Hint**: You might find that making yourself an excel spreadsheet is useful here.

Lastly, though the chip specs a Vcc from 4.5 to 15V, the chip will operate correctly and well at a much lower supply voltage (anything above 2V). This is because all of the transistors have been updated in more modern versions of the chip.

APPENDIX B: USEFUL INTERRUPTS FOR READING THE SENSORS

The Uno32 has a few peripherals that are very useful for reading the sensors. This appendix will detail some of the mechanics and guide you on how to use them to read the various sensors.

CHANGE NOTIFY: The first peripheral that you will be using in this lab is the "Change Notify" subsystem. The input Change Notification (CN) 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 selected input pins. Note that this will ONLY work on pins set to input (TRISx = 1), and you have to indicate (through registers) which pins you want attached to the CN subsystem.

The enabled pin values are compared with the values sampled during the last read operation of the designated PORTx register. If the pin value is different from the last value read, a mismatch condition is generated. The mismatch condition can occur on any of the enabled input pins. The mismatches are ORed together to provide a single interrupt-on-change signal. The enabled pins are sampled on every internal system clock cycle.

When a CN interrupt occurs, *the user must read the PORTx register associated with the CN pin(s)*. This will clear the mismatch condition and set up the CN logic to detect the next pin change. The current PORTx value can be compared to the PORTx read value obtained at the last CN interrupt or during initialization, and used to determine which pin changed.

We will provide for you the basic interrupt handler and setup, but you will need to fill it in to do the appropriate thing. You will be using the CN interrupt to read the Encoder and the Ping Sensor in this lab, so you should get used to how it works. Basically, once the pins have been designated as inputs and attached to the CN system, any change in any one of the pins will trigger the interrupt. On entering the interrupt, you must read the associated PORTx register to reset the CN subsystem so that it will trigger on the next change.

TIMERS: The second peripheral you will be using for this lab is the timer interrupt. The PIC32 device family has two different types of timers, depending on the particular device (certain timers can be joined together to double the bit width). Timers are useful for generating accurate time-based periodic interrupt events for software applications or real-time operating systems.[†††††]

Timers have an input prescaler that can be used to slow down the count rate. Each timer has a register (TMRx) which keeps track of the count, and a period match (PRx) register that causes it to reset (and can cause it to generate an interrupt). A timer that uses a 1:1 timer input clock prescale, operates at a timer clock rate that is same as the peripheral bus clock (PBCLK), increments the TMRx count register on every rising timer clock edge. The timer continues to increment until the TMRx count register matches the period register (PRx) value. The TMR count register resets to 0x0000 on the next timer clock cycle, and then continues to increment and repeats the period match until the timer is disabled.

Timers using a timer input clock prescale = N (other than 1:1) operate at a timer clock rate (PBCLK ÷ N), and the TMRx count register increments on every Nth timer clock rising edge. For example, if the timer input clock prescaler is 1:8, the timer increments on every eighth timer clock cycle. The timer continues to increment until the TMRx count register matches the PRx period register value. The TMRx count register

[†††††] Note that timers can do much more than count the system clock; for a full description of the various timer modes refer to the PIC32 Family Reference Manual.

then resets to 0x0000 after 'N' more timer clock cycles, and then continues to increment and repeats the period match until the timer is disabled.

Timers can be used in myriad ways, but essentially they allow you to control precisely when the next interrupt occurs. The way we will be using the timers is to set the prescalar to get the best "resolution" for our application, and then manipulating the period match (PRx) register to produce an interrupt as appropriate.[‡‡‡‡‡] You will be using timers with the ping sensor and the TIMERS software module uses one to give you microsecond timing.

INPUT CAPTURE: The last peripheral you will be using for this lab will be the input capture (IC). The Input Capture module is used to capture a timer value from one of two selectable time bases on the occurrence of an event on an input pin. The Input Capture features are useful in applications requiring frequency (Time Period) and pulse measurement.

The input capture works by automatically copying the value of the timer register (TMRx) into a buffer on the event occurring (specific high-low transitions on a designated pin). This makes it very easy to read the time period of a pulse coming into the pin.[§§§§§] There are only two timers that can be used with the input capture module, and while you can "double dip" and have the timer interrupt also being used for something else, generally you leave the underlying timer running at a fixed overflow interval.

For a full description of the Input Capture subsystem, see the PIC32 Family Reference Manual (Ch. 15). The system is fully configurable to generate low-high-low timing or high-low-high timing (or many other possibilities). You can even have it only trigger on every 4 or 16 events (in effect, implementing some averaging in hardware). As long as the underlying timer has not rolled over more than once between events, simple subtraction using unsigned integers will give you the correct result of elapsed time.

In this lab, you will be using the Input Capture to time the period of the square wave of the relaxation oscillator implemented using the LM555 and your capacitive touch sensor.

[‡‡‡‡‡] For example, the Uno32 runs a peripheral bus clock of 40MHz, and the ping sensor needs a 60msec delay before triggering again. Given that the timers are 16-bits, a 1:1 prescalar would give you a max rollover time of 1.34msec. You need a prescalar of 1:64 to be able to get 60msec, which means each tick of the timer is 1.6µs.

[§§§§§] Note that there are very few input capture pins available on the micro, and that they share the same timers that can be used for PWM, making them even more of a scarce resource.