Lab #2: Encoder, Ultrasonic, and Capacitive Touch

ECE167: Sensing and Sensor Technology
University of California Santa Cruz

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 angle and then drive the color as appropriate. The ultrasonic distance 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 ultrasonic distance sensor, and using various techniques to measure the change in capacitance, along with developing 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 Nucleo-64 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.

Warning: *this lab is going to be a large step increase in work over the previous two. Read the documents provided very carefully 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 the lab. As a general rule of thumb: time preparing is worth 3-5× time in the lab without preparation. Do the prep work!*

Hardware needed

Nucleo-64 + IO shield, speaker, audio amplifier (and potentiometer for volume), breadboard, rotary encoder, ultrasonic distance sensor, capacitive touch board, LM555 timer chip, assorted capacitors and resistors.

Part 0: Setting up the code structure

Within your Lab2 git folder, please use PlatformIO to create one project module for each part of this lab: QEI (for Part 1 code), PING (for Part 2 code) and CAPTOUCH (for Part 3 code).

Part 1: QEI

Incremental or quadrature encoders (QE) are used to measure angular displacement. 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 distance track of axles on CNC machines and tracking tank turret angles.

They are called quadrature encoders because the signal 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 when the other is either rising or falling, you can generate a direction in addition to a count (or accumulated angle). Quadrature encoders 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 encoders have an index mark that pulses once per revolution to give you a reference angle.[1]

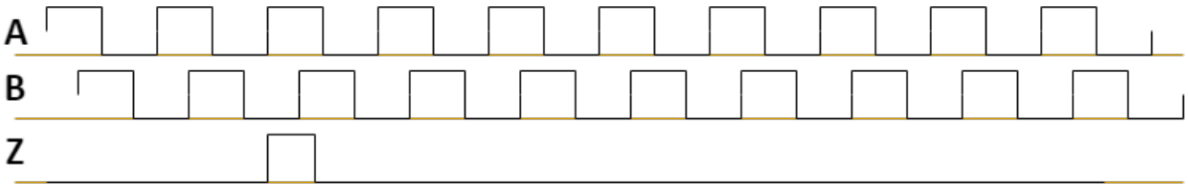


Figure 1: Sample Quadrature Encoder Pulses

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. The STM32 possesses a Quadrature Encoding Interface (QEI) subsystem that implements the state machine in hardware, but for the sake of learning you are asked to implement the state machine yourself.

The STM32 has external interrupts 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. Inside your project directory, will be implementing the QEI state machine inside a file you create called `QEI.c`. It must follow the specification outlined in the `QEI.h` file located in your Common directory–you cannot change this header. The header file contains relevant implementation details in the comments and should be studied carefully. You will have to use external interrupts. 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.

[1] 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.

1.1 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:
https://cdn.sparkfun.com/datasheets/Components/Switches/EC12PLRGBSDVBF-D-25K-24-24C-6108-6HSPEC.pdf

You should be able to find a test circuit diagram. Consult with the TA if you have difficulty. 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. 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.

1.2 QEI Software Implementation

You will be using GPIO interrupts to generate your count. Each time the interrupt is generated, you need to 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. `QEI_ResetPosition()` should reset the count (module variable) and the state of the QEI state machine.

1.3 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. Using your readings from the encoder, you will map the output angle to a color wheel, as shown below in Figure 2. 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. This specific encoder produces 24 pulses per revolution on each channel (A and B), and thus can generate a resolution of four times that (or 96 counts per revolution). This is done by triggering your QEI state machine on every transition of the encoder signals.[2]

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). Some of them are quite simple. To see some ideas, check out the Color Spaces article on Wikipedia.

[2] By triggering on every edge of the “A” or “B” signal, you are 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.

Figure 2: Color Wheel

Part 2: Ultrasonic Distance Sensor

The second sensor used in this lab is a time-of-flight ranging sensor that uses ultrasonic waves. Time-of-flight sensors work by having a transceiver inject a signal into the environment and measure the time it takes for the signal to propagate to an object, reflect (echo) off the object, and then return back to the transceiver. Time-of flight sensors are not limited to ultrasonic signals–you can use any sort of propagating wave, be it magnetic, RF, sound, or visible light. These kinds of sensors are used widely in a variety of applications, from mm ranges (vertical cavity side-emitting lasers, or VCSEL, and other diode lasers) to 100s of kilometers (weather RADAR). To measure the propagation times, techniques range across simple timers, phase locked loops, and interferometric techniques. 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: $d = 0.5 * v * t$ (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 generates an ultrasonic sound wave (~40KHz) way above the range of 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 work well.

From the timing diagram in the datasheet, cdn.sparkfun.com/assets/b/3/0/b/a/DGCH-RED_datasheet.pdf, you can see there are several things happening that lend themselves to a state machine triggered by timing and the GPIO interrupts of the STM32. 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. 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 that the sensor may not generate an output pulse if the device cannot generate a valid echo timing.

2.1 Capture Ultrasonic Distance Sensor Output

Set up the software to control the ultrasonic distance sensor to measure distance. The basic idea is to trigger the ultrasonic distance sensor every 60 msec using a timer. You will use GPIO interrupts and the µs timer (`timers.h`) to infer the distance based on the elapsed time between the trigger and when the echo pulse is detected by the sensor.

A simple state machine consisting of two states is recommended to implement the code. Your state machine should: trigger the sensor for 10μs, then wait 60ms to start the ultrasonic distance cycle again. During this waiting state you will time the arrival of the echo and calculate distance. You will be implementing this in non-blocking, interrupt-driven code within a file you create called `PING.c`. Again, your code must implement the functions specified in `PING.h`. The comments in the header file will be helpful.

To trigger the sensor, set the trigger pin to high and use a timer to generate an interrupt 10µs later (using the `ARR` register). In the interrupt, lower the trigger pin, thus ensuring that you have held the trigger up for at least 10μs and configure the timer to start the trigger sequence after 60 msec (again using the `ARR` register). During the waiting state, the rising and falling edges of the echo signal will trigger an interrupt. Record the microsecond time (`timers.h`) of these events and calculate the pulse width[3]. With this value calculate the distance measurement (avoid using any floating point arithmetic if possible!).

Your code should be able to output distances to serial so you can read them. Take measurements with known distances (use a ruler or similar), and record the time it takes for the ultrasonic signal to propagate to the wall and back, 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 ultrasonic distance 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 ultrasonic distance sensor working to the TAs (including the serial output).

2.2 Calibrate the Ultrasonic Distance Sensor Using Least Squares

The datasheet gives you a basic equation for distance using the specified 340m/s speed 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 get a more accurate sensor.

Least squares 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. 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 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 (measured using a ruler, tape measure, etc.) and y_i is the output of the ultrasonic distance 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

[3] 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.

complete range of distances you are interested in. Also note that the units of y_i don't matter–it could be in meters, μs, or counts–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, $x = (y - b)/m$. Thus the best estimate of the distance x is using the best estimate of those parameters m and 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:

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{Y}} = \underbrace{\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} m \\ b \end{bmatrix}}_{\mathbf{p}}$$

where the data pairs are inserted into the matrices Y and X for all n measurement pairs, and p is a matrix containing our parameters m and b. The solution to the matrix form of the equation above that gives the least squares estimate $\hat{p}$ for m and b is:

$$\hat{p} = \begin{bmatrix} \hat{m} \\ \hat{b} \end{bmatrix} = [X^T X]^{-1} X^T Y$$

These are called the *normal equations* and their derivation is beyond this lab. Note that Matlab doesn't implement least squares this way due to numerical instability of the normal equations. In Matlab, use: *p=X\Y* and extract *m* and *b* from the vector *p*.

Finally, for this lab, create for yourself a solid target (metal, foamcore, etc.) 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–take a few measurements per distance. 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 normally distributed? 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.

2.3 Tone Out Based on Distance

Now that you are able to calculate distance with the ultrasonic distance 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 ultrasonic distance sensor.

You should consider having a 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 (< 3−4 ft) so that you can have something like a Theremin. Demonstrate this to the TAs and include the code in your lab report.

Part 3: 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. 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. We are using projected capacitance by using 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 and unstable). This change in capacitance is what you will be measuring, and there are various ways to do this measurement.

3.1 RC Time Constant Based Measurement

The first way to measure capacitance is using the RC time constant, τ. The time constant is the time it takes for the voltage across the capacitor to rise from 0 to 63.2% of the value of the applied voltage[4]. It is a function of the resistance and capacitance: $\tau = RC$. To begin, place your capacitive touch sensor and a large (greater than 100K) resistor in series to make a low-pass filter. Drive this circuit a square wave created by a signal generator. We suggest starting with a drive frequency of 2kHz, then observe the output of the low pass filter, and adjust the drive frequency such that the capacitor has enough time to fully discharge before the next drive period begins.

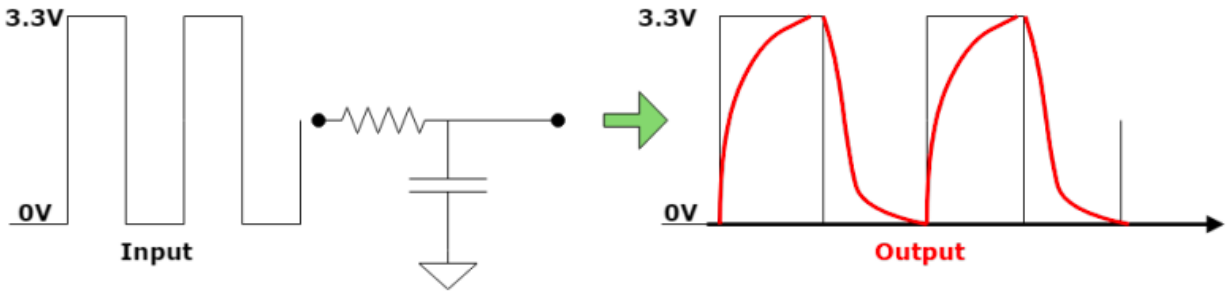


Figure 3: RC Circuit for Capacitance Sensing

Observe how the output changes when 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.

3.2 Capacitive Bridge and Differential Amp

By using a capacitive bridge and differential amplifier, you can create a more stable capacitance measuring system. By pitting two low-pass (or high-pass) filters with known resistor/capacitor values in

[4] 62.3% is derived from the fact that $0.623 \approx 1 - e^{-1}$, and $V_c(t) = V_{in}(1-e^{-t/\tau})$

a bridge configuration, then adding the variable capacitor in parallel with one of the fixed capacitors, and finally sending the outputs through a differential amplifier, you can get rid of many of the noise sources. This still requires driving the bridge with a square wave, and has additional hardware in the form of a differential amplifier, but has many advantages over the simple RC time constant measurement. *NOTE: the op-amps below in Figure 4 are actually differential and/or in-amps that you will use two or more op-amps to construct.*

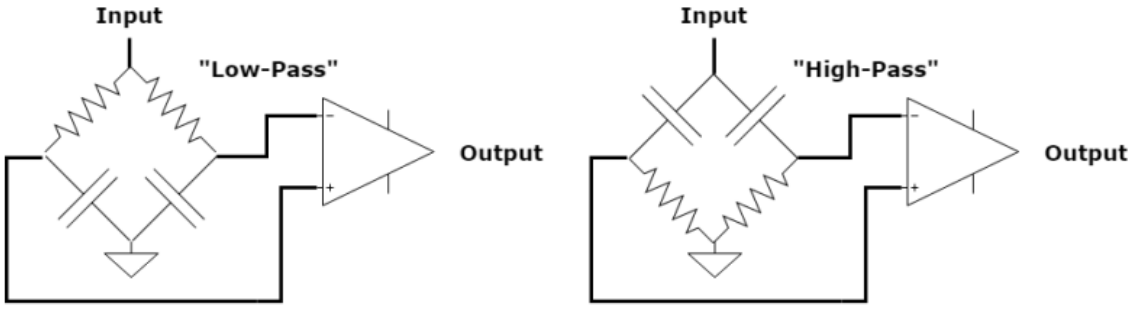


Figure 4: Capacitive Bridges for Capacitance Sensing

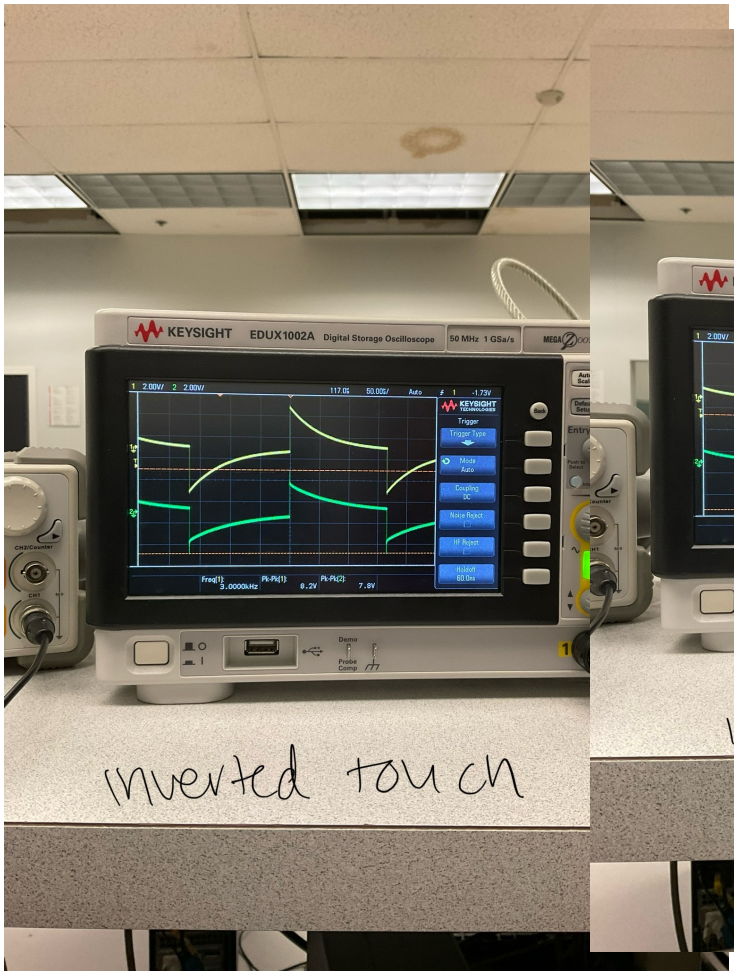
There are several ways to construct this circuit, and in this section of the lab you are going to experiment with them to discover which is the best to use.

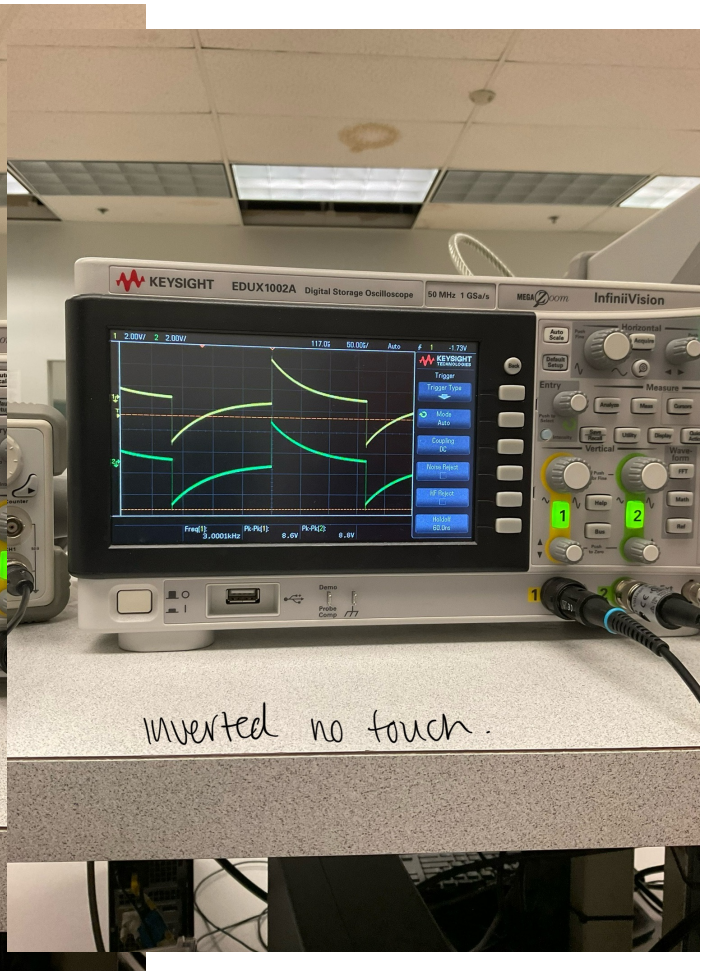
Create the first bridge with the resistors on top and the capacitors on the bottom (use approximately 100pF capacitors). Place the touch sensor in parallel with one of the capacitors, and drive the top of the resistors with the signal generator. You may use the math capability of the oscilloscope to 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 differential amplifier using the MCP6004 op-amp with appropriate gains to get a usable signal. Verify with the oscilloscope and the signal generator that you are getting the output you expected. Show this to the TAs, and include your schematics and the oscilloscope traces in your report.


KEYSIGHT EDUX1002A Digital Storage Oscilloscope
50 MHz 1 GSa/s
inverted touch


KEYSIGHT EDUX1002A Digital Storage Oscilloscope
50 MHz 1 GSa/s
InfiniiVision
inverted no touch.

3.3 Relaxation Oscillator

This final capacitance measurement system uses an astable multivibrator. You may have noticed that the best frequency of the driving square wave might change, depending on the capacitance of your sensor. Using a multivibrator, you can create a variable frequency square wave depending on the capacitance. The LM555 in astable mode implements a relaxation oscillator, which essentially runs the output of the low-pass through two comparators into an S-R latch (flip-flop) that is used to create the driving square wave. Note: See Appendix A for more details on how to set up the astable mode. The relaxation oscillator thresholds are ⅔ and ⅓ V_{cc}.

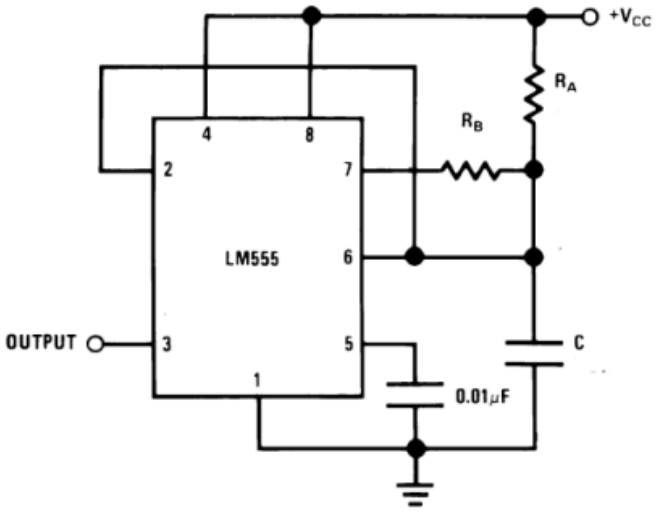


Figure 5: Example 555 Oscillator Circuit

For C in the diagram above (between threshold pin 6 and ground pin 1), put a fixed 22pF capacitor in parallel with the touch sensor. For the resistors, make sure the values chosen are such that the base capacitance oscillates at a frequency that is reasonable (within the 1-5KHz 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 STM32. You are now ready to implement the `CAPTOUCH.c` functionality that will use an external GPIO interrupt configured on the rising edge to measure the period of the incoming square wave. Again, read the comments of the header file, `CAPTOUCH.h`, carefully as many of the specifications are there. 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 external interrupts and Timers, see Appendix B.

3.4 Capacitive Touch Software Interface

For the capacitive touch, you are going to use input capture to measure frequency (period). Recall that a relaxation oscillator creates a variable frequency (but constant duty cycle) square wave into the MCU.[5] 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.

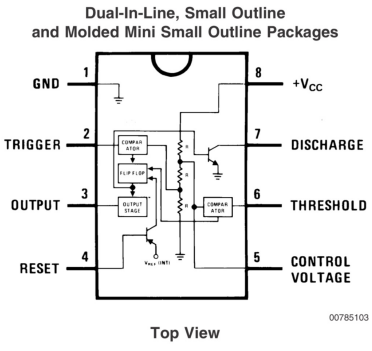
You will find that when using the capacitive touch sensor the signal is extremely noisy and will require filtering (averaging) to get a usable "touch/no touch" indicator. There is a tradeoff between the number of samples to average (more latency to touch) vs signal noise. Play with this to get a good sense of where the sweet spot is for your setup. Again, keep your frequencies on the relaxation oscillator lower than about 5KHz (you can adjust this with your resistors on the LM555).

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.

Part 4: Check-Off and Lab Report

Ensure that all of the pieces below have been checked off to the TAs and be ready to explain your methodology if asked.

Your lab report must describe your methodology and explain your results thoroughly enough such that any student who has taken this class would be able to reproduce your work. Explain how you did this lab, what challenges you observed and overcame, and make sure to answer any questions posed throughout this lab manual. What were your transfer functions for the sensors? How many measurements did you take? What were they? How accurate is your sensor model? What is your error margin?



$$f = \frac{1}{T} = \frac{1.44}{(R_A + 2R_B)C}$$

Figure 6 may be used for quick determination of these RC values.

The duty cycle is:

[5] 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.

Lab 2 Rubric

- 40% check off (10 point scale)
    - Demonstrate encoder can count up and down, show that `QEI.h/.c` works as required. Encoder should not lose counts when being moved, max should roll over to 0. (2 points)
    - Demonstrate the color change of RGB LED based on encoder counts. (1 point)
    - Demonstrate ultrasonic distance Sensor (accurate to better than 1cm); show that ultrasonic `PING.h/.c` works as required. (2 points)
    - Demonstrate tone from distance. (1 point)
    - Demonstrate capacitive RC change on oscilloscope. (1 point)
    - Demonstrate capacitive bridge change on oscilloscope and in software. (1 points)
    - Demonstrate LM555 Relaxation Oscillator on oscilloscope and in software. `Capacitive.h/.c` works as required. (2 points)
- 15% code quality (comments, readability)
- 15% code compiles
- 30% lab report
    - General format followed; all sections of lab thoroughly addressed (15 points)
    - Writing detailed enough for replication (10 points)
    - Image quality and writing quality (grammar, spelling, appropriate tone) (5 points)

Acknowledgements

Figures 1, 3, 4, and 6 courtesy of Gabriel Hugh Elkaim (using draw.io);
Figure 2 from https://commons.wikimedia.org/wiki/File:RGB_color_wheel_24.svg;
Figure 5 courtesy of TI at https://www.ti.com/lit/ds/symlink/lm555.pdf

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 ⅔ and the other at ⅓ 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.

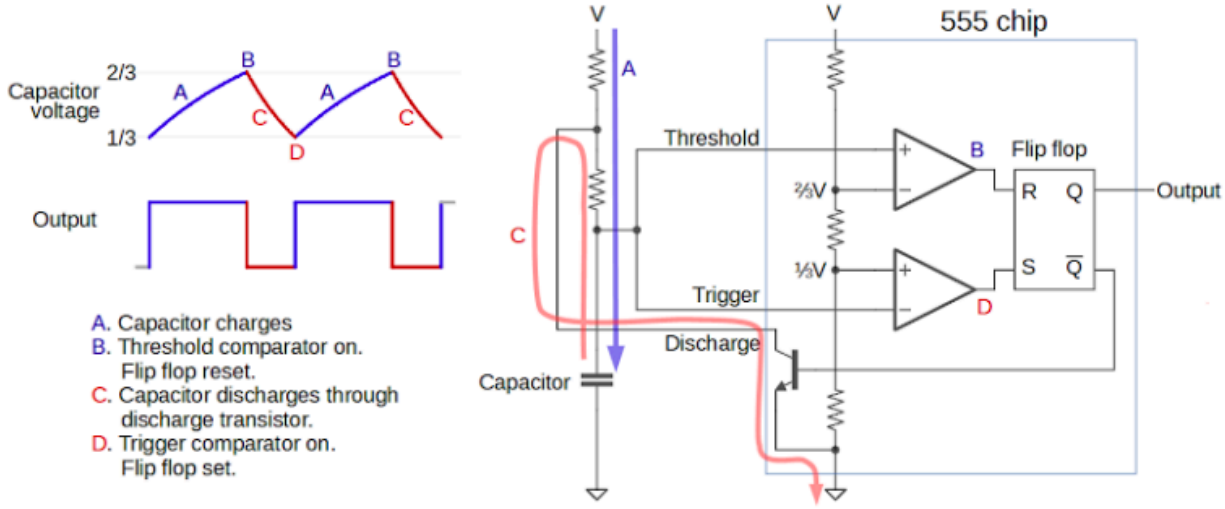


Figure 6: Conceptual Operation of 555 Timer

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, 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. You might find that making yourself a spreadsheet is useful here.

Lastly, though the chip specs a V_{cc} 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 STM32 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.

Timer Interrupts

Table 4. Timer feature comparison

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complemen- tary output	Max. interface clock (MHz)	Max. timer clock (MHz)
Advanced -control	TIM1	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	Yes	100	100
General purpose	TIM2, TIM5	32-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	50	100
	TIM3, TIM4	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	50	100
	TIM9	16-bit	Up	Any integer between 1 and 65536	No	2	No	100	100
	TIM10, TIM11	16-bit	Up	Any integer between 1 and 65536	No	1	No	100	100

The STM32 provides seven general purpose timers (see above). All these timers have a 16-bit resolution, meaning they can count to 65535 before rolling over to zero. The only exception is TIM2 and TIM5 which are 32-bit. We have reserved this timer for use in the `timers.c/h` library since counting microseconds typically requires more than 16-bits. These general purpose timers can technically count as fast as the peripheral bus clock (84 Mhz for STM32F411) but in most cases you want to count slower than this. Enter the prescaler! Prescaler is a piece of hardware that divides the clock source by some specified amount. Say we wanted a 1 Mhz timer, we would set the prescaler to 83 (84 Mhz / (84 -1) = 1 Mhz). The

reason you need to subtract one from this value is because the prescaler is zero based. That is, a prescaler of 0 means divide the clock by 1. We can generate an interrupt when the timer reaches some desired value. This is done at initialization by setting the period field to a value between 0-65535. You can also change this value at runtime by modifying the `ARR` register associated with the timer. For example, if we were using TIM3, we would modify the period by doing `TIM3->ARR = x`. Below is an example initialization of TIM3.

```
   TIM_HandleTypeDef htim3; // must be module-level or global
   // this block inits the timer generated interrupt
   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
   TIM_MasterConfigTypeDef sMasterConfig = {0};
   htim3.Instance = TIM3;
   htim3.Init.Prescaler = 83; // divide by 1 prescaler (84-1) = 1 Mhz tick
   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
   htim3.Init.Period = 999;
   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
   htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
   if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
   {
       return ERROR;
   }
   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
   if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
   {
       return ERROR;
   }
   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
   if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=
HAL_OK)
   {
       return ERROR;
   }
   HAL_TIM_Base_Start_IT(&htim3); // start interrupt
```

This code above initializes TIM3 with a prescaler value of 83 such that it counts at 1 Mhz or every microsecond. The interrupt is configured to occur every 1000 ticks since the period field is set to 999 (remember it's zero based) which means the timer will cause an interrupt every millisecond. Also note that at any point you can access the current value of the timer using the `CNT` register. For example, `uint32_t val = TIM3->CNT`. When an interrupt occurs, the MCU will halt normal execution and switch to executing code in the interrupt service routine (ISR). Below is the framework for TIM3 ISR. Remember that anything in this function should be fast, no function calls except for debugging!

```
// TIM3 ISR
void TIM3_IRQHandler(void) {
   if (__HAL_TIM_GET_IT_SOURCE(&htim3, TIM_IT_UPDATE) != RESET) {
       __HAL_TIM_CLEAR_IT(&htim3, TIM_IT_UPDATE); // clear interrupt flag

       // your code
   }
}
```

GPIO Interrupts

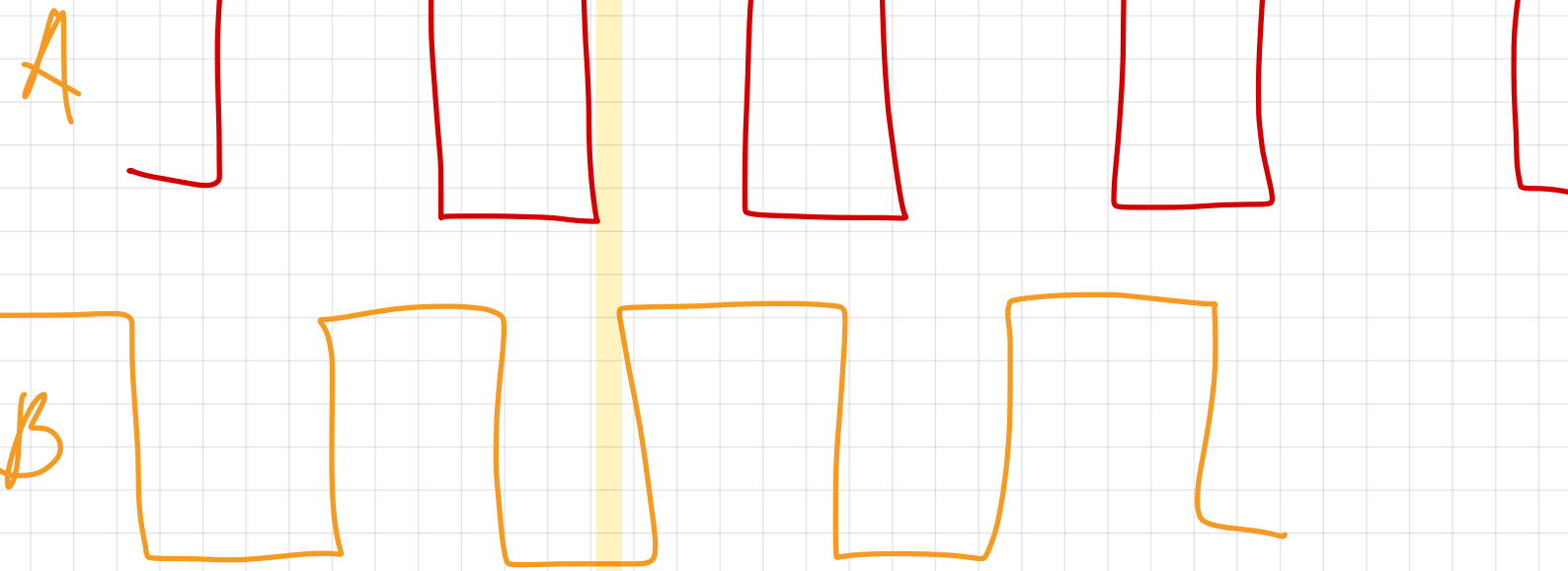
The STM32 supports interrupts that are driven by changes in the state of external GPIO pins. Below is an initialization example for an external interrupt.

```
   GPIO_InitTypeDef GPIO_InitStruct = {0};
   GPIO_InitStruct.Pin = GPIO_PIN_5;
   GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING_FALLING;
   GPIO_InitStruct.Pull = GPIO_NOPULL;
   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
   HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
   HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
```

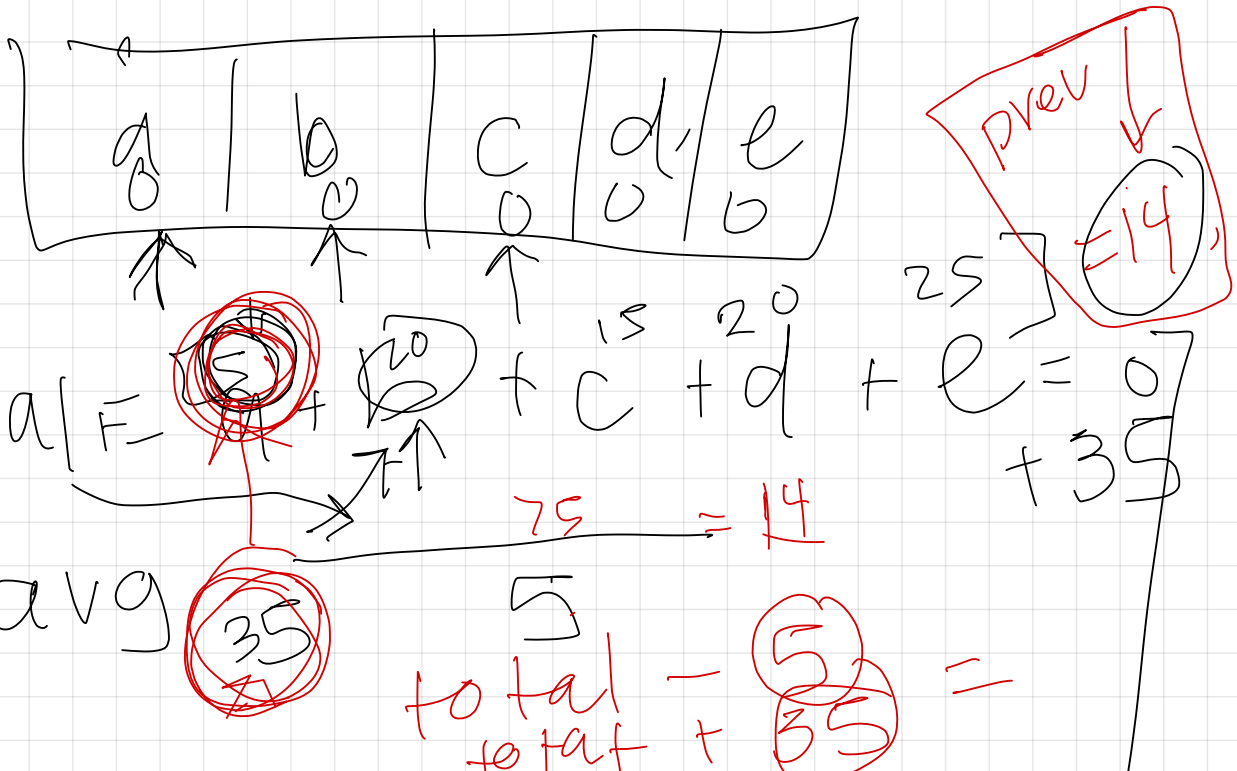
This code block sets up pin PB5 to cause an interrupt on both the rising and falling edges. One can also configure the interrupt to occur on only the rising or falling edge by setting the Mode field to `GPIO_MODE_IT_RISING` or `GPIO_MODE_IT_FALLING` respectively. Below is the framework for an external interrupt ISR. Note that the name of the ISR (`EXTI9_5_IRQn`) is based on which pin is configured to cause the interrupt, the best way to know what interrupt declaration to use is to look at the generated code through STM32CubeMX.

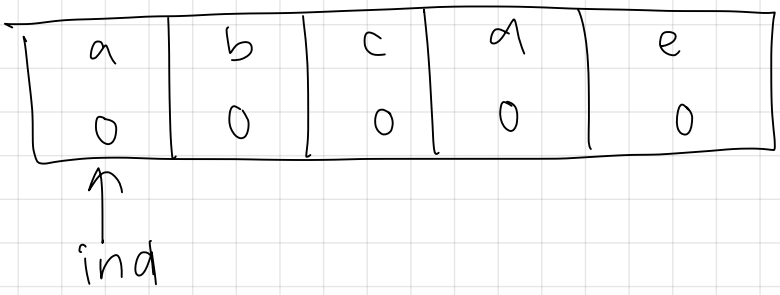
```
void EXTI9_5_IRQHandler(void) {
   // EXTI line interrupt detected
   if(__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_5) != RESET) {
       __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_5); // clear interrupt flag

      // your code
   }
}
```


A
B

f


a
b
c
d
e
prev
=14
25
total = 5 + 20 + c + d + e = 0
15 20
+35
75 = 14
5
avg
35
ind=0
total - 5 =
total + 35



$$total = [\overset{5}{a} + \overset{10}{b} + \overset{15}{c} + \overset{20}{d} + \overset{25}{e}] = \frac{75}{5} = 14$$

$$avg = \frac{}{5}$$

[0] index

① Prev = 14

② new currT = 35 , ind = 0

total = [~~5~~ + 10 + 15 + 20 + 25]

sub old −5

add new +35

[35 + 10 + 15 + 20 + 24]

③ update index

④ calc new avg total = 105

$$avg = \frac{}{5} = 21$$

end: prev = 14 avg = 21

differenal gets compared

total = 35 + 20 + 15 + 25
                10

total = 105

avg = 105 / 5 = 21

21 > MAX

prev = avg

curr time

total switch

↳ sub old

↳ add new

index

update

calcate avg

1. set prev to avg
2. get curr time