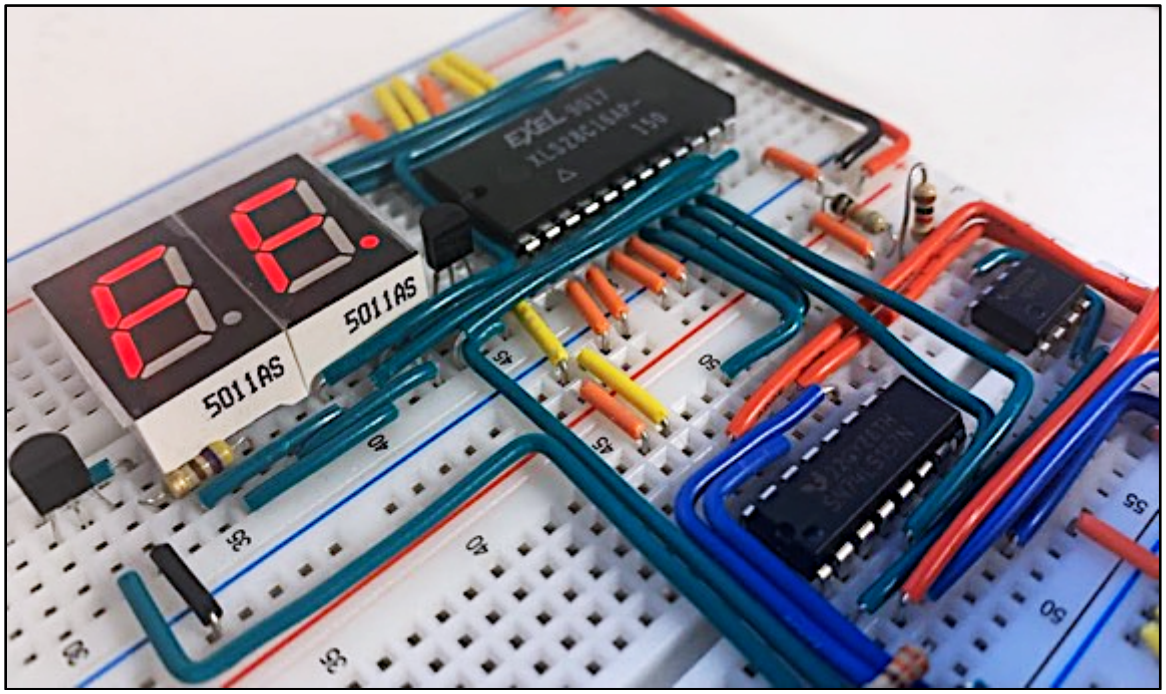
# Design Engineering Report




Course: Computer Engineering
Code: ICS2O, ICS3U, ICS4U
Author: Liam McCartney
Date: June 5, 2024

# Table of Contents

# Project 1.1 Voltage H-Bridge

## Purpose

The purpose of this Voltage H-Bridge is to demonstrate the usage of potentiometers to divide voltage and to show some of the applications that this function of potentiometers can be used for. In this analog circuit, a potentiometer is used to change which direction the current is flowing through a colour-changing (bicolour) Light Emitting Diode (LED).

## References

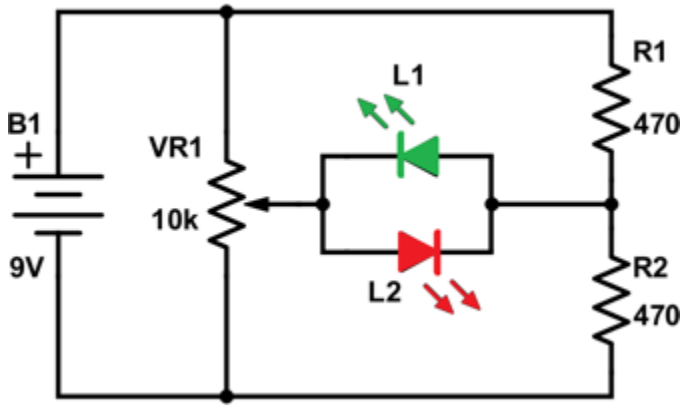
Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#VoltageHBridge

## Procedure

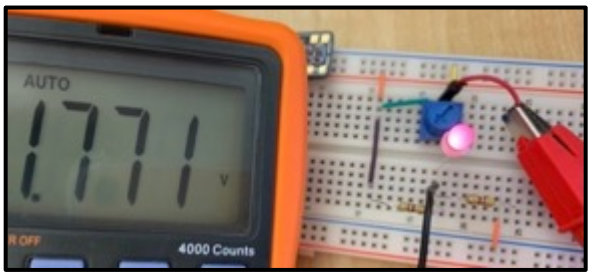
The Voltage H-Bridge is an analog circuit where a Bicolour LED changes colours when a potentiometer is turned. The parts needed are listed on the right. Take note that the two brown wires and two of the yellow wires are not part of the circuit but are instead used to connect the positive and negative lines on either side of the breadboard. Therefore, they might be missing from some pictures of the circuit. The essence of how this circuit works is simple. The Bicolour LED being used will change colour between red and green depending on which direction the current is flowing through it. The potentiometer, or variable resistor, controls how much voltage is allowed into the LED and the direction it comes from. Because this circuit has more than two states, , it is analog (as opposed to digital). The LED can be many different brightnesses of green or red and can also be off in an analog circuit.

| **Parts Table** | |
|---|---|
| Quantity | Description |
| 1 | 9V Battery |
| 1 | ACES DC Breakout Board |
| 1 | 10k Ω Potentiometer (Variable Resistor) |
| 2 | 470 Ω Resistor (Fixed) |
| 1 | Bicolour LED (5mm) |
| 8 | Total Assorted Jumper Wires |
| 1 | Orange Wire (0.4") |
| 3 | Yellow Wire (0.5") |
| 1 | Green Wire (0.6") |
| 1 | Purple Wire (0.8") |
| 2 | Brown Wire (1.1") |

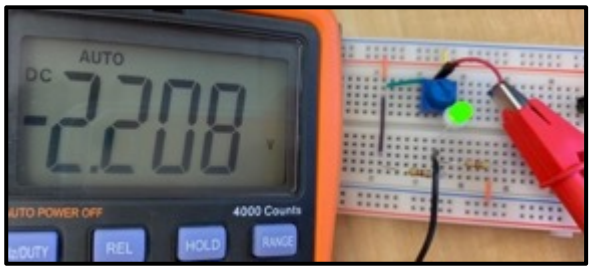
With the schematic on the right, it can be seen how this circuit works. In essence, if you turn the arrow on VR1, the potentiometer, you can change the colour of the LED. The resistors also play important roles which will be explained later on with specific examples of the circuit with the light being red, green, and off. On a basic level, they are there to provide resistance to prevent the light from burning out and also help change the current direction.



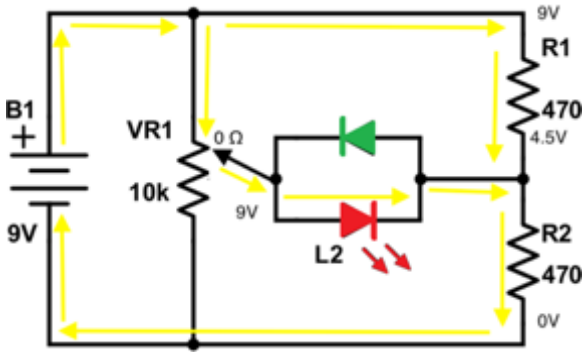
Here it can be seen that depending on the positioning of the potentiometer, the voltage changes and therefore the light changes colour. The reason that the voltage is negative in the second picture is that the current is now flowing in the opposite direction, not because it is negative. It means that the current is flowing from the black wire to the red wire as opposed to red to black which is the normal direction.



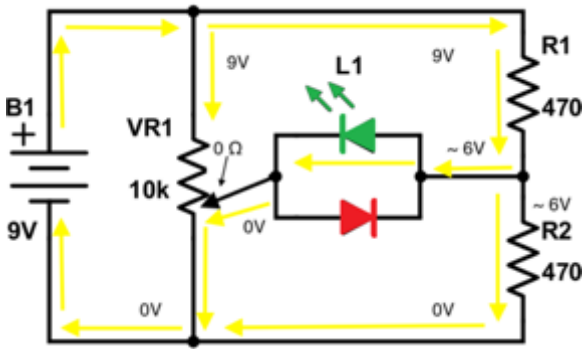

The wires were purposefully not changed between pictures because the negative voltage helps to understand that the current is now going in a different direction. It should also be mentioned once again that the wires being used to connect the power and ground line on either side of the board are currently out of view in these shots, but the two sides are connected and therefore, the orange wire at the bottom of the circuit is connected to



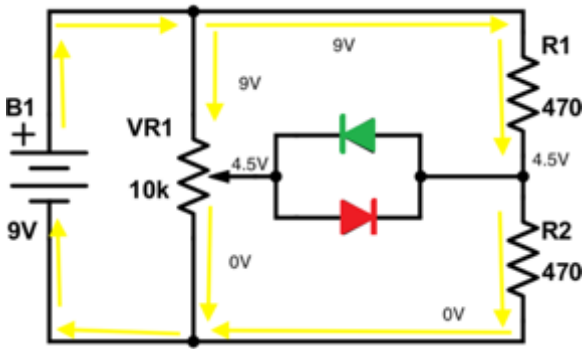

The schematic here has had more information added to it and it shows the LED in the red state. When the potentiometer is in this position, there is 9V of potential going into the light, compared to the 4.5V after the first 470 Ω resistor. This creates a 4.5V difference between the two, causing the electrons to flow through the light which takes up about 3V, still leaving 6V compared to 4.5V (a 1.5V potential difference for current to flow). Finally, both parallel paths connect and go through the last resistor resulting in 0V as they connect to ground. Because the current is flowing from left to right, the LED is on red (this has to do with what orientation the LED was put in the circuit in). As seen below, if the current were to flow the other direction, the LED would turn green.



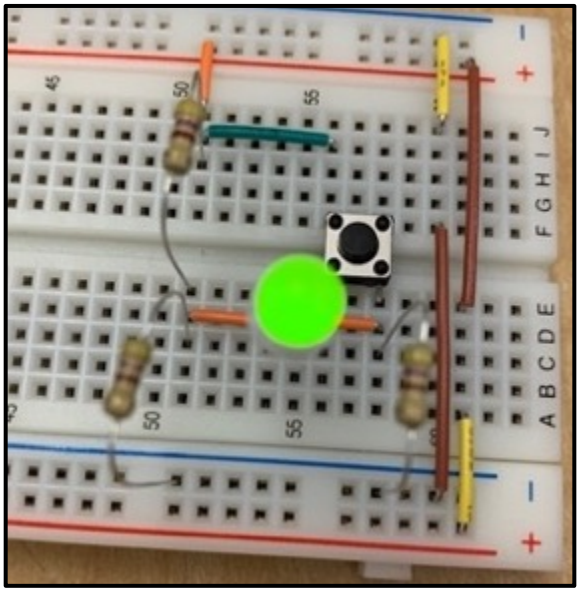

This schematic shows the light in a green state. Although it looks very complicated, while the light is in a green state most of the circuit can be discounted. What is happening here is that the current goes through the first 470 Ω, then goes through the light from right to left making the light green, and then goes to ground. None of the other components affect the light. The 10k Ω potentiometer takes 9V from the current going straight down and offers 0 Ω to the current coming through the LED, so it has no effect. All that is happening is the LED is attached to a 470 Ω resistor and then to ground, only with a lot of confusion around it.

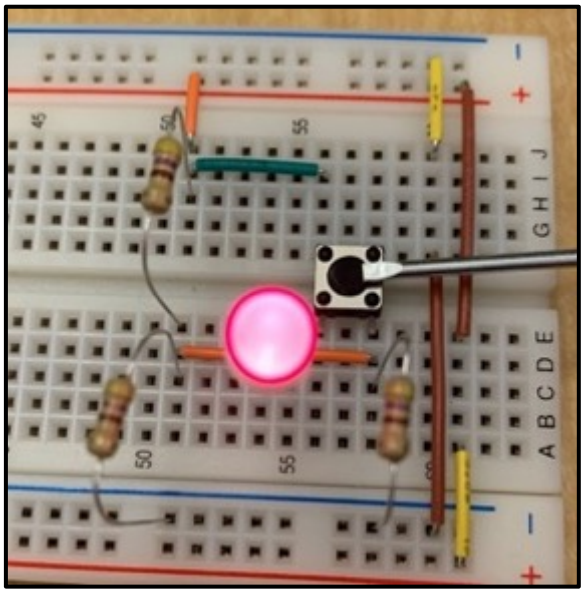


Finally, we see the LED in an off state. This is because there is no difference in voltage when the C lead of the potentiometer is halfway in the middle of A and B. There is 4.5V of potential on either side of the light, so there is no flow. The current simply skips the light, goes through the potentiometer and the two fixed resistors, then goes back to ground. Even if the C lead is not exactly in the middle, the light would still be off. There needs to be 3V available for the light to turn on, and as seen in the picture under the schematic, the light will not turn on with small amounts of power. In the picture, the potentiometer is positioned almost exactly halfway in between red and green, and there is only a 3.4mv (0.0034V) of potential difference in the circuit.









The circuit seen on the right is the same circuit but digital instead of analog. This means that the light only has 2 states, bright red and bright green, whereas the analog circuit from earlier had many different shades of each colour and could also be turned off. The way that this circuit works is also simple. Currently, as seen on the right, when the button is unpressed the light is green. When the button is pressed (bellow) the current can now flow through the button with a resistance of 0 Ω, and that makes it the current change direction. With the button unpressed, the current would flow from the top left resistor, through the LED, and then into ground through the bottom right resistor. When the button is pressed, the current instead flows through the button (because it has no resistance), through the LED, and then into ground through the bottom left resistor

## Media

Project Video: https://youtu.be/_Um4IbwVVAc

## Reflection

In my opinion, this project went well. I think I did well on time management, I got all the pictures necessary for the document done by Thursday and began to write the report on Friday night. I know that this may seem like I started a bit late, but I like to sit down and tackle a project, not take it in small steps over a long time. I feel that I'm more productive if I block off time, and work until it's done. I still have a few hours of time left while writing this reflection before the project is due, so my time management can't have gone terribly. The only reason I decided to start on Friday as opposed to getting it done earlier in the week is that I wanted to have learned what was taught about word and looking back to how I would have written this report a week ago, I think it was well worth it. Even though I wasn't short on time, I will still start earlier next time because it's impossible to foresee setbacks, and extra time is always good. Something that I found interesting about the project was comparing the different circuits that perform the same task and breaking down what each of them does to get the same result. I find it fascinating that there can be so many different solutions to one problem, and it is fun to identify what all the solutions have in common.

# Project 1.2 The Two-Bit Adder

## Purpose

The purpose of the two-bit adder is to explore the uses of logic gates and the binary number system to understand how computers do mathematical operations. The two-bit adder applies the theory of the binary system to electronics to make High and Low states which are the basics of how computer logic, and by extension computer arithmetic and mathematics, function without the base 10 decimal system.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#TwoBit
Digital Logic Simulator (Program used in screenshots): https://sebastian.itch.io/digital-logic-sim

## Procedure

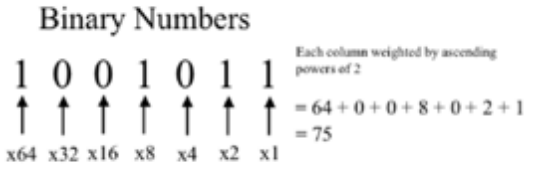
The two-bit adder is simply a network of connections between the four-input slide switches, some basic logic gates, and the three output LEDs. What makes it an adder is the way binary, high and low states and logic gates are applied to the circuit. To understand the two-bit adder, it is important to understand all of those concepts and how they work together to turn a board with wires in it into an adder with a coherent output. This circuit uses AND, OR, and XOR logic and has two smaller circuits in it. Those circuits are the full adder and the half adder, and an understanding of both is essential to an understanding of the two-bit adder.

**Parts Table**

| Description | Quantity |
|---|---|
| 9V Battery | 1 |
| Aces DC Breakout Board | 1 |
| 470 Ω Fixed Resistor | 3 |
| XOR 4070 CMOS IC | 1 |
| AND 4081 CMOS IC | 1 |
| Green LED | 3 |
| Slide Switch | 4 |
| Assorted Jumper Wires | 73 |

### The Binary Number System

As previously alluded to, binary is the number system used by computers. In the world of electronics, it is hard to have a distinct signal for each of the 10 digits in our base 10 number system. However, it is easy to have two digits, 0 and 1 in the case of binary. While every place value is either 10 times greater than or less than the digit to its left or right in decimal, that does not work in binary with only the digits 0 and 1. So instead, each place value is two times greater than the place value immediately to the right. What is the ones column in base 10 is still the ones column in binary, but whereas the 10s column is the second column to the left of the decimal place in decimal, the second column is the twos column in binary. To the left of the twos column there is the fours column, then the eights column, then 16s, and so on. For example, the number 1001 in binary is 9 in decimal, and 10101 is 21. In essence, the only difference between the systems is the difference between each column's weight when summing up every digit in a number to arrive at the total value.

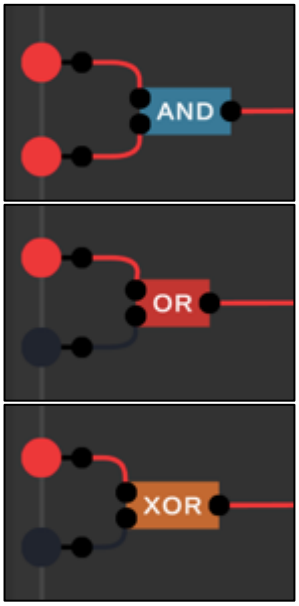


The last piece of necessary knowledge about the binary system is the bit. The word bit is an abbreviation of "binary digit". For example, a four-bit number would have four digits (bits), all of which would be either 0 or 1, and a seven-bit number would have seven digits (bits). The only unfamiliar concept of bits that is not seen in the decimal system is a bit limit. The bit limit is the maximum possible value made with a specified number of bits, for example, the two-bit limit is three, or 11 in binary. This is important because the two-bit adder adds two two-bit numbers together to form a three-bit number with a maximum possible value of six, found from multiplying the two-bit limit by two. This means that the output display will only need to be able to display a maximum value of six, meaning the adder only has to have a three-bit display because six is less than the three-bit limit of seven (111).

## High Low States and Logic

As mentioned already, electronics function using the binary system. The previous section noted that the binary system is easier to use for computers. That is because by default, electronics can have two states: high, or more than half of the source voltage; and low, less than half of the source voltage. For example, a light can only be on or off. This translates well to binary with 1 being assigned to places with high voltage, and 0 being assigned to places with low voltage. These states can also be called "yes" and "no", "true" and "false", "on" and "off", etc. Logic gates use these high and low states, or 1 and 0, to perform logic. Logic operations take (usually) two inputs of 0 or 1 and perform their logic, and then return a single output of either 0 or 1. These logic operations are what is used in the two-bit adder. The logic gates used in the two-bit adder are the AND gate, OR gate, and the XOR gate.

Truth tables can be used to illustrate the utility of a logic gate. A truth table shows what a logic gate will output in a given situation. Every logic gate has a unique truth table, although some are the inverse of other logic gates. As seen to the right, the AND gate will only output a 1 if both inputs are 1, otherwise it will output a 0. An OR gate will output a 1 if either input is a 1. An OR gate has essentially the same functionality as simply merging two separate wires, which is why the OR integrated circuit (IC) is not used in the prototype for the two-bit adder. The "exclusive or" (XOR) gate is like an OR gate, except if both inputs are 1, it will output a 0. An XOR gate will output 1 if one and only one input is a 1, otherwise it outputs 0. If all three of these gates are applied correctly, it is possible to make a circuit that will add two 2-bit numbers together.



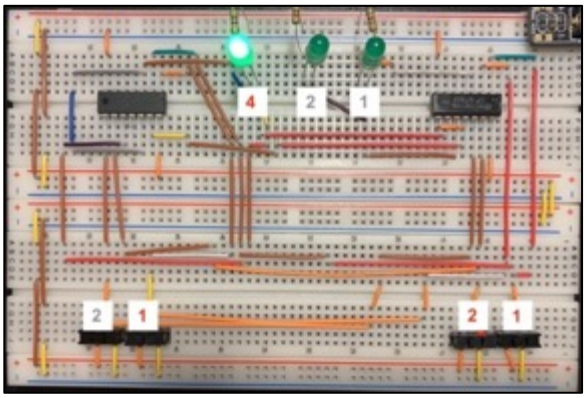

| Truth Table | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

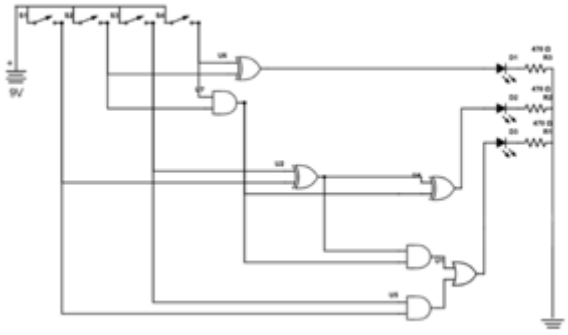
| Truth Table | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| Truth Table | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

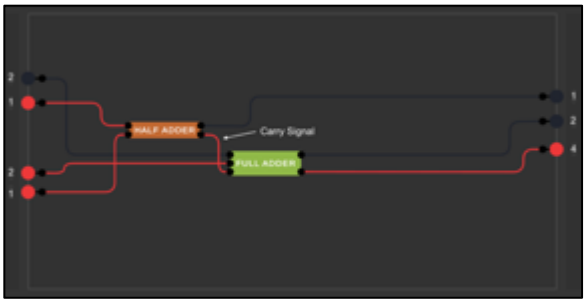
## The Circuit

What the two-bit adder does is take two two-bit numbers, chosen by turning 4 switches on or off, and runs them through logic gates to produce a three-bit output displayed with three LEDs. As shown with the red numbers in the picture, the adder is currently adding the two-bit numbers 01 and 11, or 1 + 3 in decimal. The output is 100, or four in decimal.

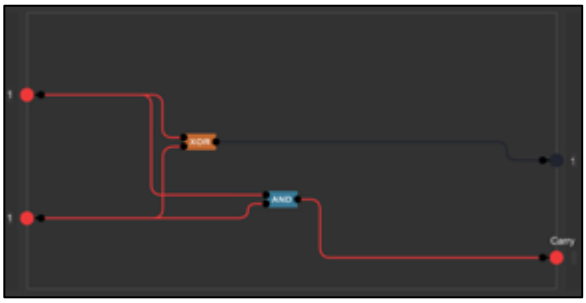




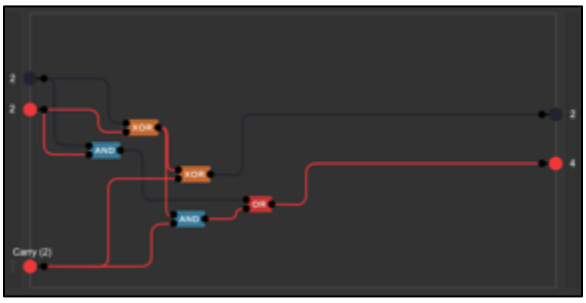
After understanding the concepts from the previous two sections, it is now possible to understand the two-bit adder (schematic to the right). The two-bit adder works by using the half adder and the full adder together. Each of those sub-sections performs "math", but their purpose is different. It is only in unison that they can produce the results that are needed. The procedure used to add numbers together is as follows: firstly, both ones digits from each two-bit number are checked. If only one ones digit is in the 1 state (switched on), the system turns on the ones digit LED. If both ones are on, then a carry signal is sent to where the twos digits are checked. There, if there is only one twos digit in a 1 state, or if there is only a carry signal present, the twos digit LED will light up. If there are two twos digits (including the carry signal) in a 1 state, the fours digit LED turns on. If all three twos digits are in a 1 state, then both the twos and fours digit LEDs will light up.




To further clarify that process, the first picture to the right is a simplified version of the schematic. It illustrates how the two-bit adder is really just the half adder and full adder. In this example, 1 is being added to 3. Both ones digits are on, so the half adder makes a carry signal. The carry signal from the half adder is then used as the carry (C) input in the full adder. The C input is treated as a twos digit and is then added with the other twos digit to get a final value of four.



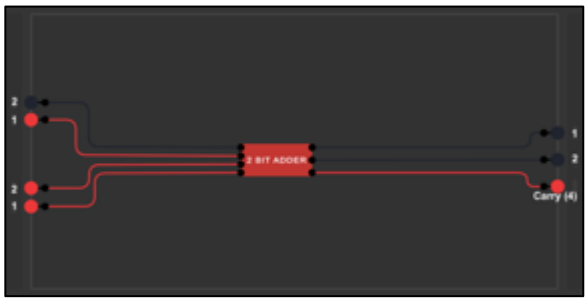

The second picture is a look at what is happening inside the half adder seen in the picture above. There is an XOR gate and an AND gate, each connected to both the ones column inputs, but each with a separate output. The XOR gate is connected to the LED representing the ones column on the display, and it is checking if only one of the inputs is on. If that is the case, that would mean the ones LED would be on because 1 + 0 = 1. However, if both inputs are on, the XOR gate does not turn on the LED. This makes sense because if both inputs were on, the half adder would be calculating 1 + 1, which does not equal one. Instead, when both inputs are on, the AND gate turns on the carry output which, as shown in the picture above, is used as the C input for the full adder.




The third picture shows what is happening inside of the full adder. The full adder's purpose is to take the two twos column inputs and the carry input, and use those to produce a output for the twos column LED and another carry output, in this case being used as the output for the fours digit LED. If the adder was adding numbers greater than two-bits, the carry output of the full adder would be used as another C input for another full adder adding the next place value. How the full adder works is similar to the half adder, just with three inputs instead of two. Firstly, the A and B inputs (the two twos place values) are fed into an XOR gate. The output of that is then led into another XOR gate with the C input, and then the subsequent output goes into the twos column LED. This process

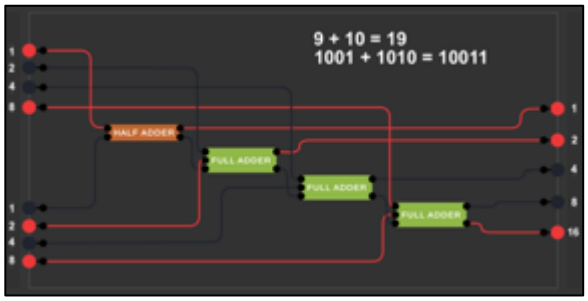
essentially checks if one and only one of the inputs are on, and if so, the twos LED comes on. However, this is not all. A and B are also fed into an AND gate, whose result goes to the fours column LED through an OR gate. The other input for that OR gate is the output of an AND gate with the inputs of C and the first XOR's output. The combined function of the whole process is to check how many of the inputs into the full adder are on. If none, do nothing, if one is on, turn on the sum output (twos column LED), if two inputs are on, turn on the carry output (fours column LED), and finally if all three inputs are on, turn on both the sum and carry output.

Combining the full adder and half adder, the result is a two-bit adder. The ones column inputs of each two-bit number go through the half adder and either turn on the ones light, turn on the carry output, or do nothing. The half adder handles the twos columns. It checks how many inputs are on and also processes the carry output of the half adder, and it controls the twos and fours column LEDs. However, adding two two-bit numbers is not very useful. Luckily, all that is needed to add larger numbers are more full adders.




## Application

Realistically, adding two-bit numbers is not useful. Two-bit numbers are so small that it's easier to just do the mental math. However, digital adders are easily expandable. The picture to the right shows a four-bit adder. All that has changed is the number of inputs, outputs, and full adders. If full adders are linked where the carry output of one goes into the carry input of another, they can form very powerful adders. Any number can be added if the simple connections are expanded in the right way.




Thinking back to the original purpose of the two-bit adder, exploring how computers complete basic mathematical operations, it becomes evident that they do not really do math. Computers doing math is an illusion. In reality, all that is happening is logic gates are taking inputs of 1 or 0, and producing outputs of 1 or 0. The adder simply tries to replicate how humans do math, with the carry output/input and a full adder for every digit, but in reality, the addition is simply a truth table. What the two-bit adder actually shows is the possibilities that 0s and 1s can present when used in the right way, because while logic gates cannot truly do math, their replication of it can still be incredibly useful.

## Media

Project Video: https://www.youtube.com/watch?v=QuX1xw3jD_g
Full Size Schematic: https://drive.google.com/file/d/1EomtwhDAFjItb0view?usp=sharing

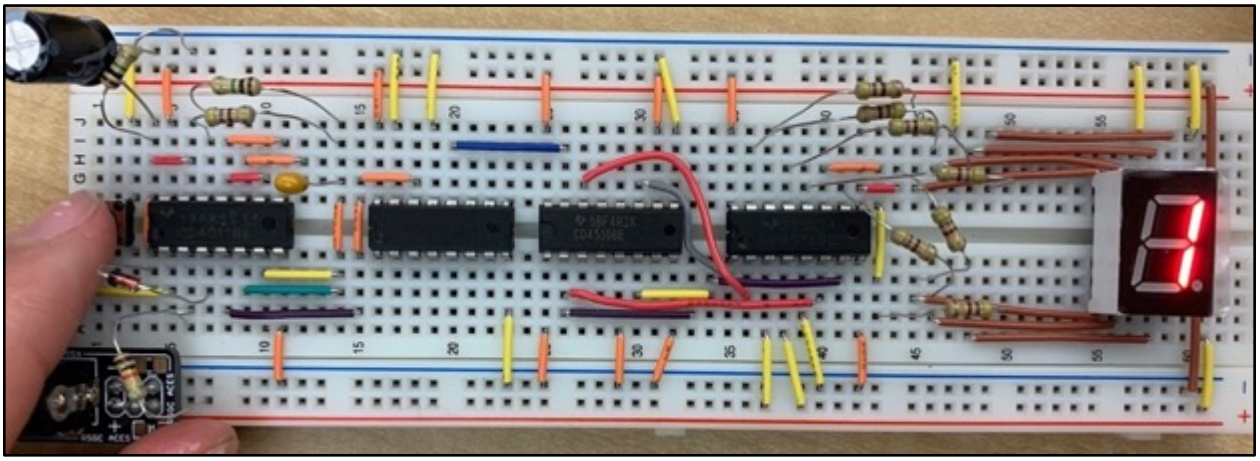
## Reflection

I think I did really well on time management on this project. This probably sounds weird because I am turning in my DER two or three hours later than last time, but that is because I had very little free time to finish the project today. I realised a few days ago, and got most of the work done before today. I planned ahead and was prepared, and didn't have to cancel any plans. One thing I noticed is that the further I get into the DER, the more I understand how the circuit works. I only really began to understood the voltage H-bridge and two-bit adder when asked to explain how they worked.

# Project 1.3 A Counting Circuit

## Theory

The counting circuit is the circuit that will be built up over the subsequent 21 pages, starting from a button input and ending with a printed circuit board with all the components soldered on, and mounted inside of a custom case for the circuit. The counting circuit uses the concept of square waves created by the analog oscillator. Their frequency and duration can be controlled by resistor-capacitor pairs which can make the counter count faster, slower, for longer, or for less time. In other words, the resistor-capacitor pairs control duration and frequency. The count is determined by using the square wave to run a 4510 binary-coded-decimal counter which is then decoded with a 4511 IC. The output from the 4511 is then used to run a seven-segment LED display, like a mini scoreboard. The circuit can also be turned on and off at any time with an on/off switch and the direction of the count is controlled by another switch. The counting circuit also introduces four new unique ICs, and three of those four are function chips, which is a new concept. Below is a preview of the circuit that is to come.



# A. Button Input

## Purpose

The purpose of the button input circuit is to start exploring square waves. The circuit takes an input from a button and extends the input, but instead of letting output fade in current it simply turns off immediately after a certain amount of time.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB
How Capacitors Work: https://www.explainthatstuff.com/capacitors.html

## Procedure

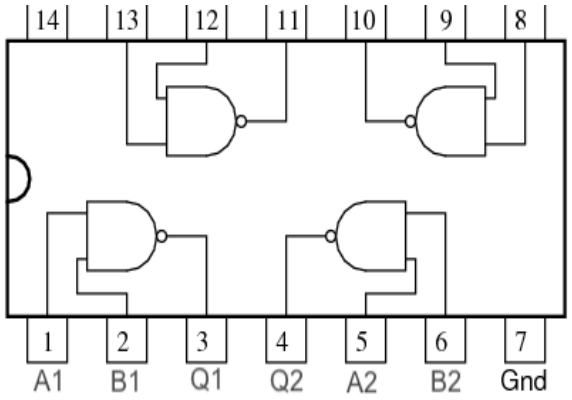
As explained in the purpose section, the button input makes a square wave. A square wave is a wave pattern, except the rising and falling edges of the wave are almost vertical lines and the plateaus are horizontal. The button input circuit has three important sections. First is the section with the button, 10k Ω resistor, and the first NAND gate. Secondly, there is the 10 uF capacitor and the 1M Ω resistor. This section is also called RC. Finally, the last section consists of the second NAND gate and the LED.

| **Parts Table** | |
|---|---|
| Description | Quantity |
| 9 V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| 1N4148 Diode | 1 |
| 10 kΩ Fixed Resistor | 1 |
| 1 MΩ Fixed Resistor | 1 |
| Button (PBNO) | 1 |
| Green LED | 1 |
| Capacitor (10 - 1000 µF) | 1 |
| Aces DC Breakout Board | 1 |
| Assorted Jumper Wires | 7 |

This circuit depends on NAND logic and capacitors, which are both new concepts in this report. NAND stands for Not AND, and as seen to the right, has the inverse outputs of AND logic. The output is always high except if both inputs are high, in which case the output is low.

| **Input** | | **Output** |
|---|---|---|
| A | B | NAND |
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

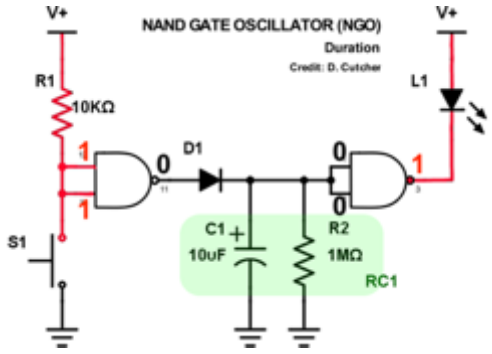
To the right is the pinout of a 4011 NAND gate. A pinout is a little diagram that shows what each pin on an IC does, because sometimes they are not self-explanatory (but this one is). The Vdd and Gnd stand for power and ground respectively. Both A and B are inputs. There are two inputs per gate, and the output of each gate is C. The numbers after the letter are the number of the gate that the corresponding pin is part of. There are a total of four NAND gates in the 4011 IC.



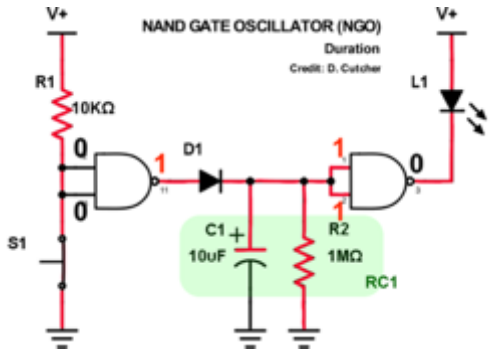

By definition, anything that can store electric charge in an electric field is a capacitor. A capacitor is simply two conductive surfaces separated by an insulator. However, this definition would technically define clouds and the ground as a capacitor. In electronics, a capacitor is two sheets of metal separated by a thin dielectric (insulator). The details of how this creates capacitance are confusing, but the knowledge

needed to understand this circuit is basic. When one side of a capacitor is charged, it stores that charge. When that side of the capacitor is then connected to ground, it will release its charge back into the circuit and travel to ground. The amount of charge a capacitor can hold is called its capacitance and is measured in Farads. For smaller measurements, micro-Farads (uF) are used. There are 1,000,000 uF in one Farad.

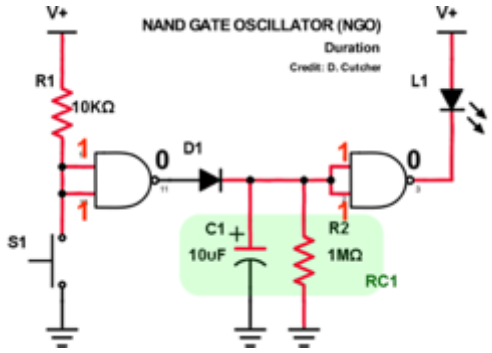
Starting in a rest state, both inputs of the first NAND gate are high, resulting in the output being low. The low output then goes into the second NAND gate, where both inputs are low and therefore the output is high. Since the positive lead and negative lead on the LED are both at 9V, there is no potential for current to flow and the LED stays off. If the led rotated so that the positive lead was connected to the NAND output and the negative lead was connected to ground, the LED would be on at rest.




The second picture shows the circuit when the button is pressed. The current that was previously going into the inputs of the first NAND gate is now going through the resistor and then straight to ground instead. Since the inputs to the NAND gate are now both 0, the output is 1. This charges the capacitor, which is important for the next stage. Now, since the inputs of the second NAND gate are both 1, the output is 0 (ground). This means the LED now has direct ground access and turns on.

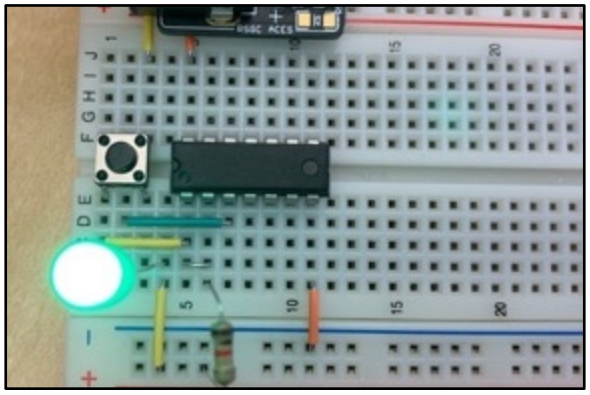



What has been described above is simply an overly complicated PBNO. The difference is that when the input button is released in this circuit, the LED will stay on for an amount of time controlled by RC1. When the button is released, the capacitor stops being charged and the current stored attempts to make its way to ground. The diode prevents the current from going into the output of the first NAND gate, so its only remaining route to ground is through the 1M Ω resistor. Since the 1M Ω provides so much resistance, it takes a long time for the capacitor's charge to fully drain. The length of time can be adjusted by changing the components in RC1. A higher resistance resistor and a higher capacitance capacitor will each make the LED stay on for longer. The time the LED stays on after releasing the button, or the pulse length, is completely customizable by changing the values of RC1. In the particular schematic above, the LED will stay on for roughly eight seconds.

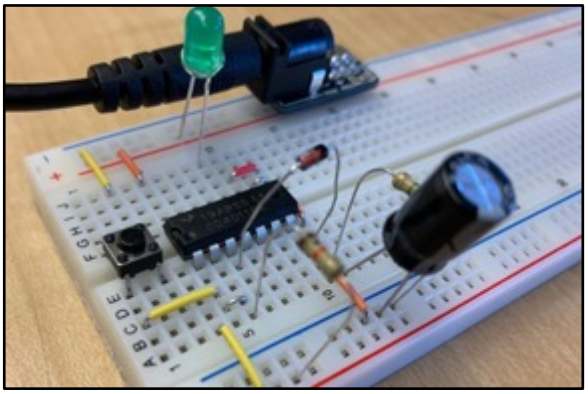



In an analog circuit, the LED would fade out until the capacitor had completely discharged, but in this circuit, the LED will immediately turn off. As already known, a high state and low state are over half the supply voltage and under half the supply voltage respectively. So according to the NAND gate, the two inputs coming from the capacitor discharging have a logic level of 1 until they do not, in which case they have a level of 0. Also, the NAND gate can only output a 1 (supply voltage), or a 0 (ground). So when the discharge of the capacitor reaches under half the supply voltage, the output of the NAND gate immediately switches from 0 to 1 and the LED turns off almost instantly with no fade. This is the beginning of a square wave.

## Media



Button Input



Button Input With RC1

## Reflection

I definitely noticed a difference when writing this report compared to my other ones because of the additional seven that also need to be done by the end of the cycle. I still tried to put in as much time as I could and stay on a good pace to stay on top of my work. One thing I did try was to get the circuit to work using NOT gates instead of NAND, but I was unsuccessful and stopped before I wasted too much time. However, I do think that this circuit could be recreated with NOT gates, I just believe I made a mistake somewhere that I did not notice. Since I had no way to prove it worked, I left it out of the report although I still found the idea interesting.

## B. NAND Gate Oscillator

### Purpose

The purpose of the NAND gate oscillator is to create a sustained square wave and explore how the frequency and duration and pulse length of the wave can be controlled with the power of resistor and capacitor pairs.

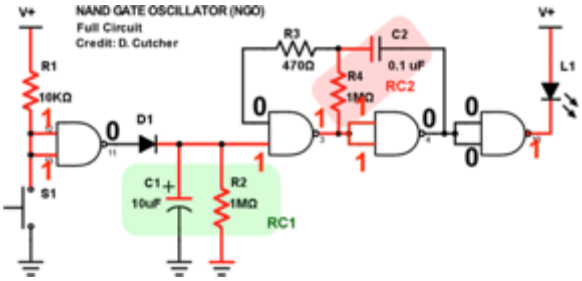
### References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB
4011: http://darcy.rsgc.on.ca/ACES/Datasheets/CD4011BC.pdf

### Procedure

The NAND gate oscillator, and all subsequent circuits in this project, are all modified versions of the previous circuit, and in this case, the button input. The NAND gate oscillator takes the single square wave generated by the button input and creates a sustained square wave with customizable frequency and pulse length. When the button is pressed, the circuit starts creating the wave, and ends it when the timer (button input length) runs out, immediately stopping the wave. In this circuit, the square wave is used to make a LED turn on and off, and with some adaptations can make a bicolour LED flicker from red to green. This circuit has no new concepts, but their applications are different than before.

| **Parts Table** | |
|---|---|
| Description | Quantity |
| 9 V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| 1N4148 Diode | 1 |
| 10 kΩ Fixed Resistor | 2 |
| 1 MΩ Fixed Resistor | 2 |
| 470 Ω Fixed Resistor | 1 |
| Capacitor (0.1-0.01 µF) | 1 |
| Capacitor (10 - 1000 µF) | 1 |
| Green LED | 1 |
| Button (PBNO) | 1 |
| Aces DC Breakout Board | 1 |
| Assorted Jumper Wires | 13 |

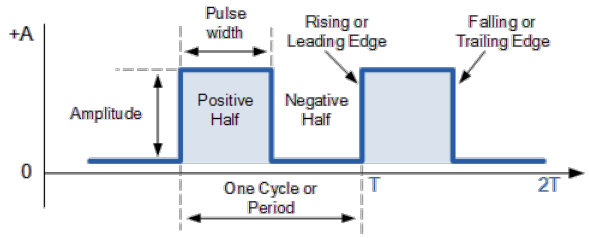



To the right, there is an annotated schematic. It depicts where we left off, with the first resistor and capacitor (RC) pair creating a single pulse square wave after pressing the button and letting go, but now the rest of the oscillator has been added. RC1 goes into the second NAND gate which does NAND logic and makes the output 1. That output then goes into a 1M Ω resistor and then charges a 0.1 uF capacitor. In the schematic to the right, the capacitor is still charging so the current is not going into the 470 Ω resistor. However, once the capacitor is done charging, the current will flow through the resistor into the other input to the second NAND gate, making both inputs a 1 and therefore the output turns to 0. This means the capacitor stops being charged and the top input into the NAND gate is relying on the capacitors discharge to keep the input a 1. Once the capacitor discharges to the point where its output is less than half the source voltage, the second input of the NAND gate turns to a 0 and the output turns to a 1. This starts the feedback loop all over again.

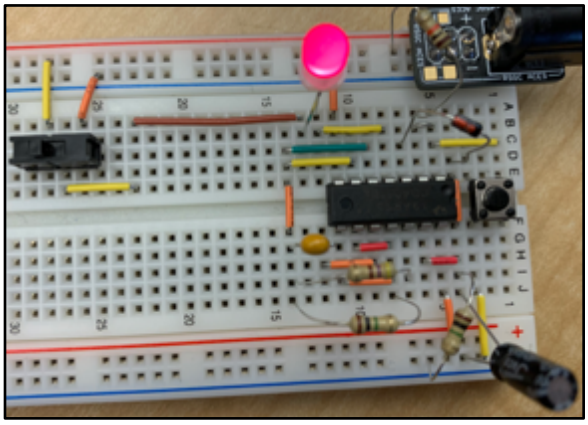
The other two NAND gates act as NOT gates control whether or not the LED has ground access, meaning they control if the LED is on or off. They switch states when the second NAND gate switches states, so the LED's negative lead will have access to ground when the output of the second NAND gate has an output of 0 (meaning the LED is on). When the second NAND gate switches its output to 1 and subsequently so does the fourth NAND gate, the LED has no potential difference between its two leads and immediately turns off. Because this process is reliant on the threshold of half the supply voltage, there are only two states for the LED: full brightness, and off. That is the point of a logic oscillator.

## Square wave

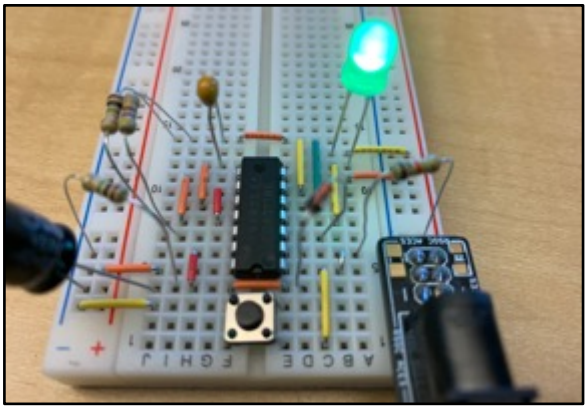
What has been created is a square wave. A square wave is a waveform (a line graph with time as the x-axis and something like voltage as the y-axis) where the rising edge and falling edges are almost verticle lines and the negative halves and positive halves are equal in time. In the case of the NAND gate oscillator, the square wave is measuring the change of voltage from 9v – 0v over time.



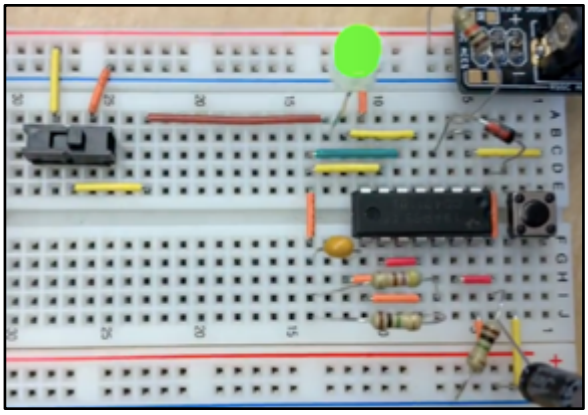

To demonstrate that the output of the circuit is oscillation, meaning that it is changing from ground to supply voltage, a bicolour LED is being used. The bicolour LED is attached to a switch, and that switch is connected to both ground and V+ depending on which way it is switched. Switching the switch will change whether the LED is on or off at rest and whether the LED blinks green or red (i.e., changes which way the current is flowing through it). If the LED was attached to both ground and V+ at the same time, the oscillator would make it flicker between red and green, meaning there is always an output of 0 or 1 even at rest.



## Media



NAND Gate Oscillator Side View



Oscillator With Bicolour On Green

## Reflection

This report was a lot easier to write than The Button Input. Building off of past knowledge made it go a lot faster and it was easier too. For the NAND gate oscillator, I was able to get a modification to work which was vindicating after the previous attempts of the NOT gate recreation of the button input failed. On an unrelated note, I also think I did quite well on keeping the circuit compact, even though it was not an official standard. I am still a little pressed for time so I am about to start on the decade counter and hopefully get a big chunk of it done if not finish its report today.

## C. Decade Counter

### Purpose

The purpose of the decade counter is to show the application of a clock signal and its utilisation with the 4017 decade counter IC and, and how square wave clocks are used in all advanced electronics. The decade counter also teaches about duty cycles of square waves.

### References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB
4017: http://darcy.rsgc.on.ca/ACES/Datasheets/cd4017b.pdf

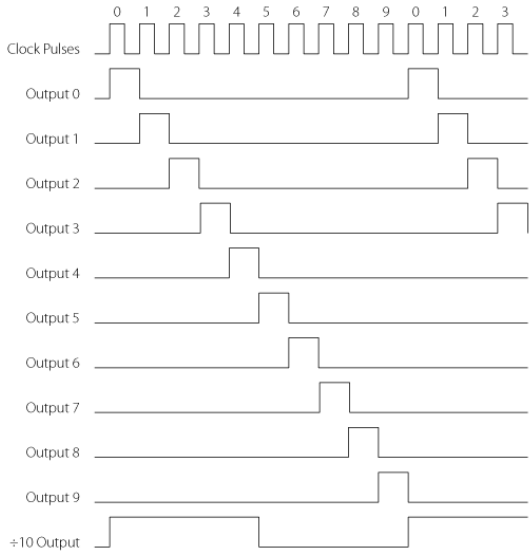
### Procedure

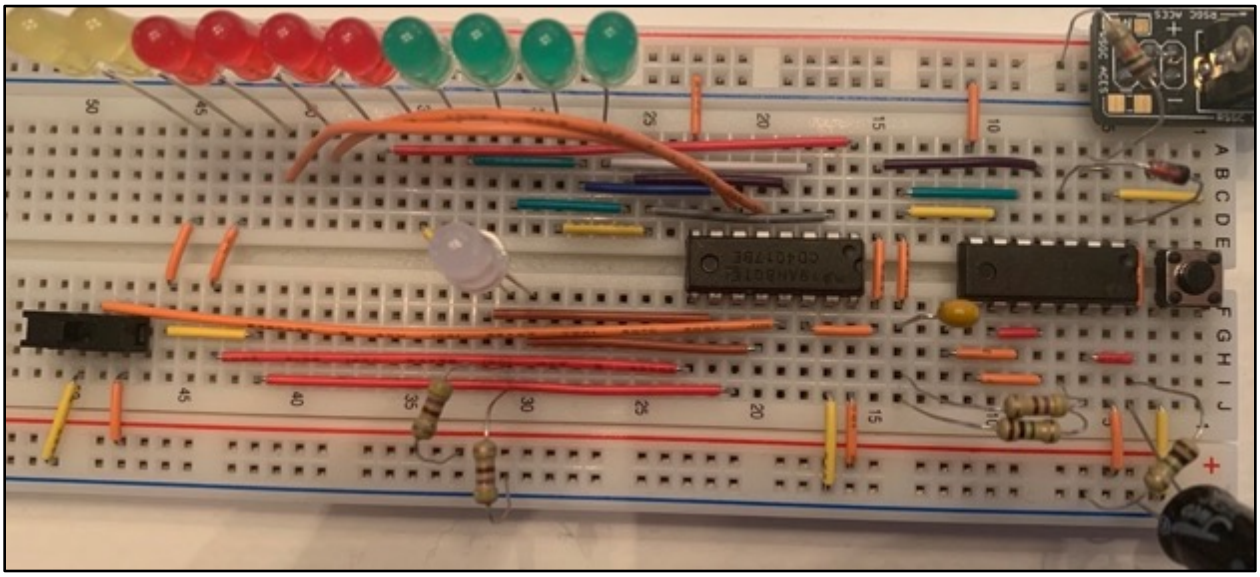
The decade counter circuit does not add any more complex logic functions to our expanding circuit. The logic is all handled inside the 4017 decade counter, the brain of this circuit. The decade counter is a 16 pin IC that counts from zero to ten. It takes an input clock cycle (which is the square wave generated by the NAND gate oscillator) and on every rising edge of the clock, it increases its count by one. It shows its count with ten outputs, each corresponding to a decimal digit. One of its purposes is to be readable by humans since it is hard for people to read binary quickly. The decade counter also has some other unique things about it. Firstly, for the decade counter to run, the clock enable (sometimes referred to as clock inhibit since the IC does not work if it has power) input must be grounded. There is also a reset functionality. If the reset pin receives power, the count will start at zero again. Finally, the ÷10 pin outputs a 0 while the count is from 0 – 4 and outputs 1 when the count is 5 – 9.

**Parts Table**

| Description | Quantity |
|---|---|
| 9 V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| Decade Counter CMOS 4017 IC | 1 |
| 1N4148 Diode | 1 |
| 10 kΩ Fixed Resistor | 1 |
| 1 MΩ Fixed Resistor | 2 |
| 470 Ω Fixed Resistor | 4 |
| Capacitor (0.1-0.01 µF) | 1 |
| Capacitor (10 - 1000 µF) | 1 |
| Assorted LEDs | 10 |
| Bicolour LED | 1 |
| Slide Switch | 1 |
| Button (PBNO) | 1 |
| Aces DC Breakout Board | 1 |
| Assorted Jumper Wires | 40 |

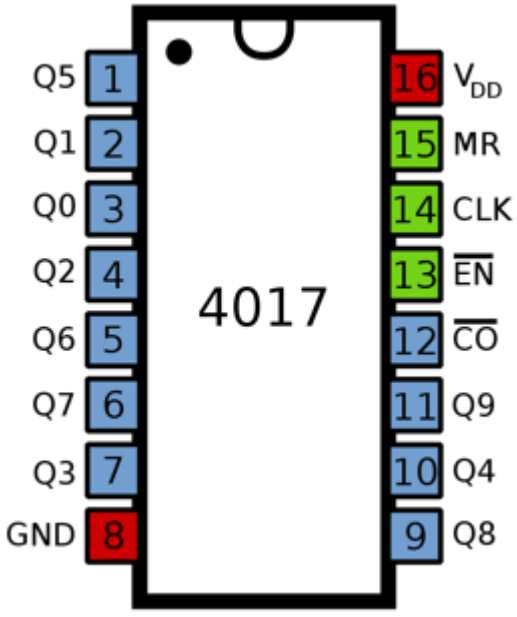
The function of the decade counter in this circuit is to turn on LEDs representing 1 – 10 in order. This is accomplished by taking the square wave from the NAND gate oscillator and putting it into the input of the decade counter IC. The IC then does some very complicated logic and produces the outputs of 1 – 10 in order. Luckily, this logic can be visualized with waveforms, this time rectangle waves. A rectangle wave is a square wave, but with a new element introduced: duty cycle. The duty cycle is the percentage of time that the wave is in a high state. Since there are 10 digits and one is always high, every digit's output is a rectangle wave with a duty cycle of 10%. The waves start in a staggered formation when the clock starts and end when the clock ends. When a wave is high, the corresponding LED lights up.





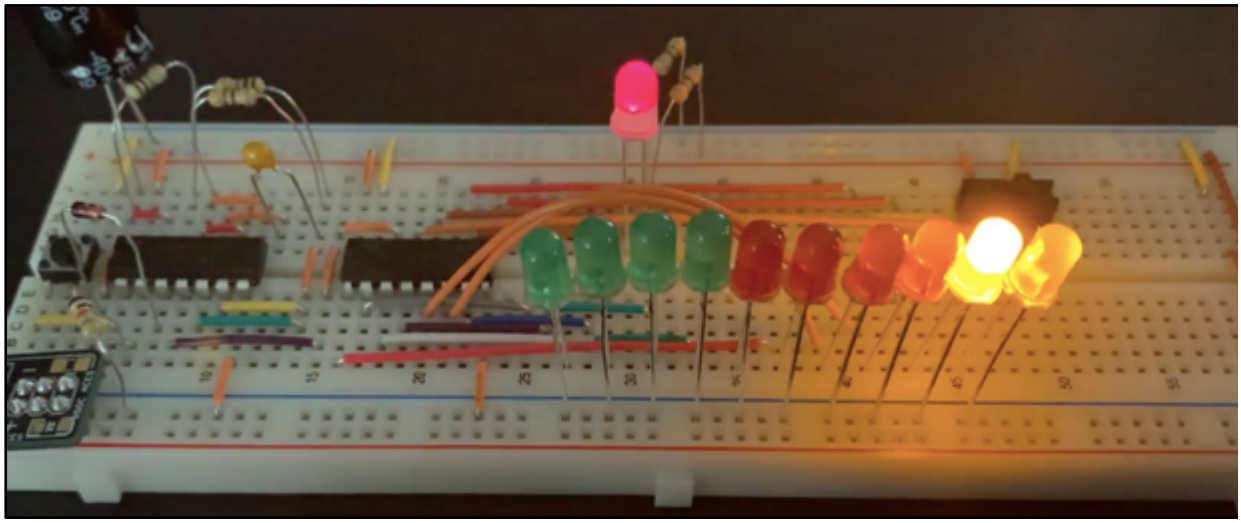
The above picture shows the decade counter circuit. The IC further right is the NAND gate and the one towards the middle is the decade counter. The display LEDs are in the top left. Some added features to this version of the circuit are a slide switch to enable/disable the clock input into the decade counter, and a bicolour LED that visualizes the ÷10 output (green when 0 – 9 and red when 5 – 9).

While the wiring might look like a confusing mess, and it is, that is due to the pinout of the decade counter (shown to the right). The numbers inside the squares are the pin numbers, so they are in order, but the numbers being represented by the pin are not in order. Pin one represents the five output and pin 10 represents four. While this would not be hard to implement on a circuit board, it is quite messy on breadboards. Any pin with a line over it means that it has to be connected to ground to function instead of the normal connection to V+ to turn on a pin.




Expanding the decade counter to count to higher numbers is quite simple. All that is needed is to connect the divide by ten output (pin 12) to the clock input of a second decade counter. By doing that, the second decade counter advances its count once for every ten times the first decade counter advances, which is exactly how the decimal number system works. Circuits that output decimal are much easier to understand and manipulate, which is one of the reasons the decade counter was designed to count decades (groups of 10).

## Media



Decade Counter in Action

## Reflection

I was very confused while writing this section of the DER because my circuit refused to work and kept skipping LEDs and was overall inconsistent. The frustrating part was every single LED lit up some of the time and they were all connected to the right output, but some would light up three times more than others. The circuit ended up working spontaneously for a few short minutes at a time without me changing a thing, but still generally was not working properly. I did some research and found that it could be due to me using a battery as opposed to getting power from the wall, and some things suggesting that I should attempt to stabilize my breadboard V+ and ground with capacitors, but nothing worked. At least I have a few clips of it working properly. I'm definitely going to try and find out the problem in class.

## D. Decimal Counting Binary Up/Down Counter

### Purpose

The purpose of the decimal counting binary circuit is to teach about binary coded decimal (BCD) and to demonstrate the uses of the 4510 and 4516 ICs, as well as to show their similarities and differences.

### References

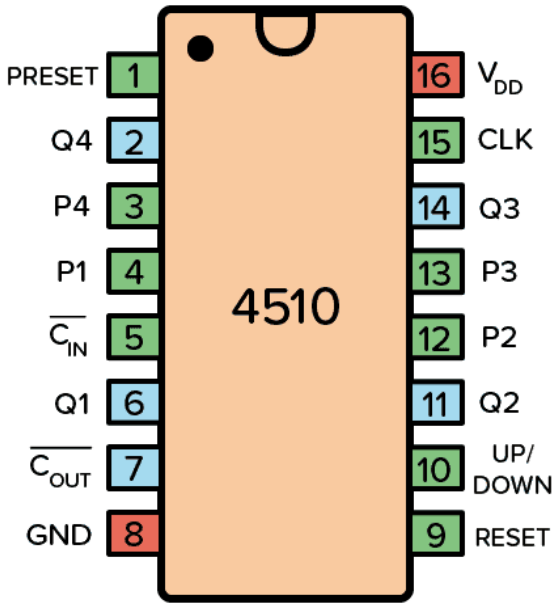
Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB
4510: http://darcy.rsgc.on.ca/ACES/Datasheets/cd4516b.pdf

### Procedure

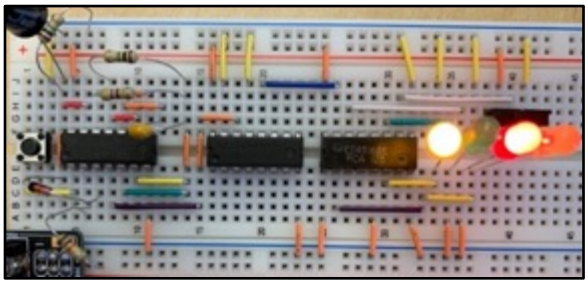
| **Parts Table** | |
|---|---|
| Description | Quantity |
| 9V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| Decade Counter CMOS 4017 IC | 1 |
| Up/Down Counter CMOS 4510 IC | 1 |
| 1N4148 Diode | 1 |
| 10k Ω Fixed Resistor | 2 |
| 1M Ω Fixed Resistor | 2 |
| 470 Ω Fixed Resistor | 1 |
| Capacitor (0.1-0.01 uF) | 1 |
| Capacitor (10 - 1000 uF) | 1 |
| Assorted LEDs | 4 |
| Button (PBNO) | 1 |
| Aces DC Breakout Board | 1 |
| Slide Switch | 1 |
| Assorted Jumper Wires | 37 |

The binary up/down counter circuit uses the previous NAND gate oscillator and part of the decade counter. It runs off of a clock cycle that is initially provided by the NAND gate oscillator and is then slowed down by the 4017. The original clock cycle goes into the 4017 and then comes out of the divide by ten output, the clock pulse now having one-tenth of the frequency. That pulse then goes into the new chip in this circuit, the 4510 (or 4516). These two chips are much like the 4017 decade counter, except they deal with binary numbers. Each chip will count to the equivalent binary number of their last two digits. For example, the 4516 counts to 16 in decimal. The chips have four output pins, meaning they can output a four-bit number (a nibble). In this circuit, LEDs are used to show when an output is on or off. These ICs can also count downwards. This output is called binary coded decimal (BCD). BCD is a term for a decimal number represented by four-bit binary numbers. However, what differentiates this from normal binary is that BCD is formatted digit by digit. For example, 156 in BCD is 0001 0101 0110 (1,5,6 in binary).

To the right is the pinout for the 4510 and 4516. They are the same. The four output pins for the four-digit output as well as the carry output are light blue. There are also some other features, such as carry-in and count down which have to be grounded to function. The clock input from the 4017 goes into the clock input. There are also input pins on the binary counter, which are not used in the circuit but are still worth explaining. The input pins are there for a situation where the circuit does not use a clock, or it isn't being used at the time. They directly affect the output. Four-bit binary numbers can be inputted into the chip as high or low voltages, and the chip will output the same number. However, this is not the main use of the 4510 and it is not the use used in this circuit. In this circuit, the chip is used to count from 0 to 10 in binary.



To the right is an image of the complete binary counter circuit. Right now, the value of the count is five (the closest LED represents the ones column). The switch behind the LEDs controls whether the up/down input (pin 10) receives V+ or ground. This changes which way the counter counts. It is important to have the switch connect to both because if the pin is left floating, the circuit becomes unpredictable.

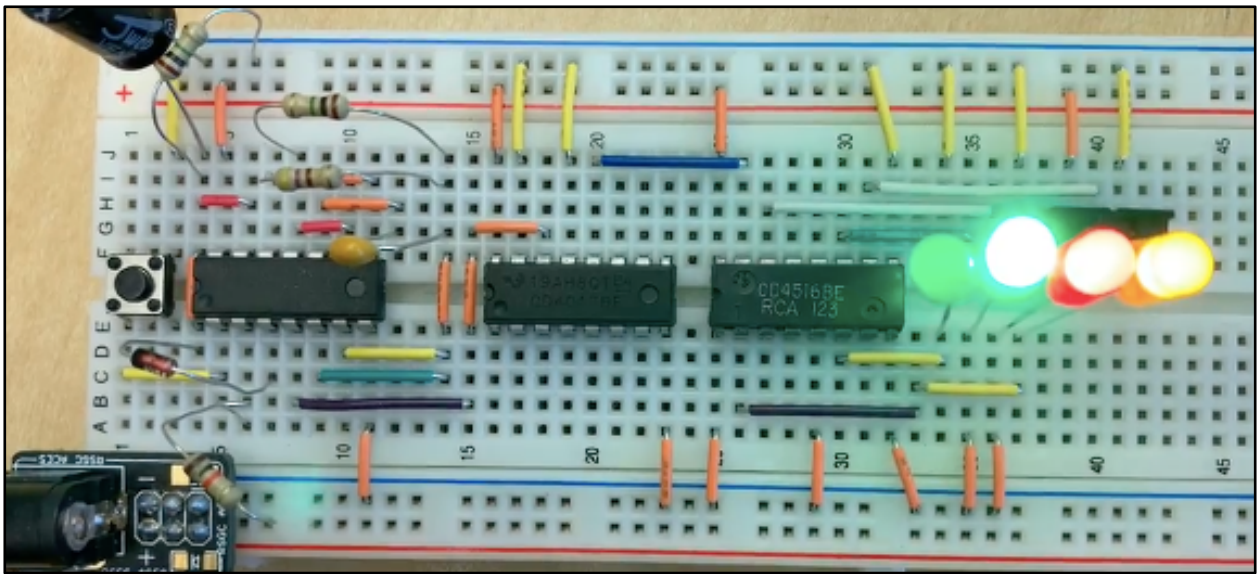


## BCD/Binary to Decimal

To the right is a table with every four-bit binary number (nibble) and its equivalent in decimal. This is the range of count with the binary up/down counter circuit. This is also where the difference between the 4510 and 4516 becomes apparent. The 4510 will only count to nine in decimal, or 1001 in binary, because it only counts ten numbers (the count starts at zero). However, if the 4516 were used, it would count up to 15, which is the four-bit limit. The reason that the 4510 was used in this circuit is that the end goal is to eventually use a seven-segment display to show the count. A seven-segment display only has ten digits, so if a 4516 were used, the display would go blank as the number of the count surpassed nine, which makes those six digits obsolete. The 4510 is more practical for applications where humans have to directly interpret the output, and the 4516 has a wider range of count for a situation where the output does not need to be easily read by people.

| Decimal | Binary |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 1010 |
| 11 | 1011 |
| 12 | 1100 |
| 13 | 1101 |
| 14 | 1110 |
| 15 | 1111 |

## Media




4510 Counter at Maximum Count of Nine

## Reflection

This circuit gave me much less trouble than the last one (which I did get to work properly in the end). It worked on the first attempt without any inconstancies. The idea of using the 4017 to slow the frequency seemed strange to me since we had spent a week on the NAND gate oscillator part of the counter where the frequency could be easily changed using resistor-capacitor pairs, but then I realised that you can alter the RC pairs and use the 4017 to get even more results, and I also know I would have missed the opportunity to explore with it had we simply used a larger capacitor to achieve the same effect of slowing the frequency. It has been two days since I wrote the last report because I have been busy outside of school so I'm working hard to get back on track and it's going well. Overall, I do not think that I am too far behind and with a bit of hard work and maybe a late-night I can easily get back to where I want to be. I should also probably say that I tried to make an adaption to the circuit like I have done in the previous ones, but I could not find anything that made sense or highlighted a concept or aspect of the BCD counter so I decided not to add more for the sake of adding more and instead chose to forgo the opportunity to add more LEDs. I think it was the right decision because I truly do believe that less quantity of higher quality work is better than pages of low-quality work (which seems ironic to say just after hitting the 10 page mark).

## E. Binary Counting Decimal Decoder

### Purpose

The binary counting decimal decoder teaches about the concept of decoding (changing a value to the equivalent value in another system) and foreshadows the potential uses of the 4511 IC to convert a nibble into an input readable by other pieces.

### References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB
4511: http://darcy.rsgc.on.ca/ACES/Datasheets/cd4511b.pdf

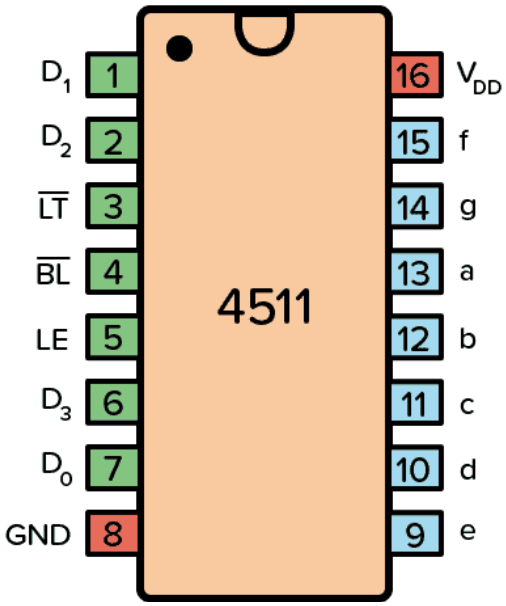
### Procedure

The binary counting decimal decoder is halfway in between the previous section (decimal counting binary up/down counter) and the next section (seven-segment display). It focuses on the 4511 IC, which is a decoder. A decoder takes information formatted in a certain way, interprets it, and then outputs the same information in a different "language" (for lack of a better word) than the original information. For example, this decoder takes in a four-bit binary number as its input and produces the equivalent output fit for a seven-segment display. While these inputs and outputs will look nothing alike, they hold the same meaning in their respective uses. The 4511 takes four inputs and turns them into seven different outputs which will later be used to run a seven-segment display; however, that is in the next section. For now, the focus is on the 4511 and its exact function.

**Parts Table**

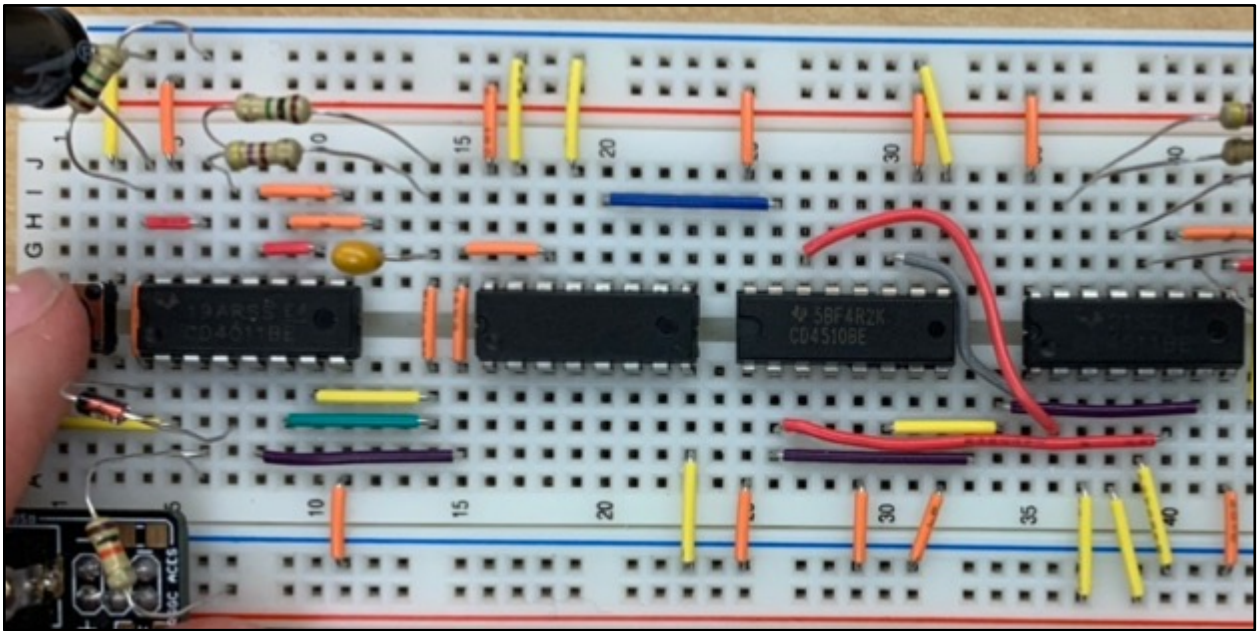
| Description | Quantity |
|---|---|
| 9 V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| Decade Counter CMOS 4017 IC | 1 |
| Up/Down Counter CMOS 4510 IC | 1 |
| BCD to Seven Segment Decoder CMOS 4511 IC | 1 |
| 1N4148 Diode | 1 |
| 10 kΩ Fixed Resistor | 2 |
| 1 MΩ Fixed Resistor | 1 |
| 470 Ω Fixed Resistor | 1 |
| Capacitor (0.1-0.01 µF) | 1 |
| Capacitor (10 - 1000 µF) | 1 |
| Button (PBNO) | 1 |
| Aces DC Breakout Board | 1 |
| Assorted Jumper Wires | 37 |

To the right is the pinout diagram for the 4511. As mentioned, there are the four inputs (labelled D1 to D4) and their respective outputs of a-g (each represents one segment of a seven-segment display). However, there are also some other pins. The LE (store), BL (blank test), and LT (lamp test) pins. When pin five, the store pin, is high, the current outputs of the IC are saved. Pin four, the blank test, will turn off all outputs when low. Finally, pin three, the lamp test, will make every output high when it is grounded. For normal use, the blank test and lamp test pins should both be connected to power, and the store pin should be grounded. If conditioned like this, the chip will function as would be expected and work as a decoder and output the equivalent of the input.



One downfall of this chip is that it is just a BCD decoder. It will only decode binary numbers and it will only give a pre-set output for every number. For example, it is impossible to feed in a binary number that goes through the decoder and comes out as just a g as its output. Or, if the user does not like the appearance of a number, they have no way to change its style (e.g., hook or no hook on a 9)

## Media




4511 IC on Righthand

## Reflection

I had some trouble separating this report from the seven-segment display since the chip is specifically made for seven-segment displays, and I added them to the breadboard in one build. I think that I probably included too much about seven-segment displays since they are supposed to be the next section and I do not want to repeat myself too much. I am going to go over this section again and try to remove mentions of seven-segment where I can, but it is necessary in some places. The build itself went smoothly I guess because all it included was connecting six wires and conditioning the pins of the 4511 correctly.

## F. Seven Segment Display

### Purpose

The seven-segment display circuit applies all the previous concepts into one circuit that counts up and down on a seven-segment display.

### References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB
7-Segment CC Display: http://darcy.rsgc.on.ca/ACES/Datasheets/5011-BS-200901A.pdf

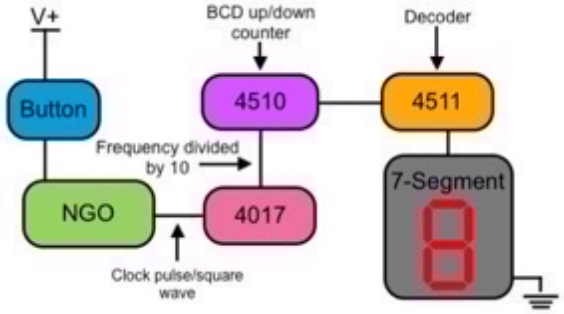
### Procedure

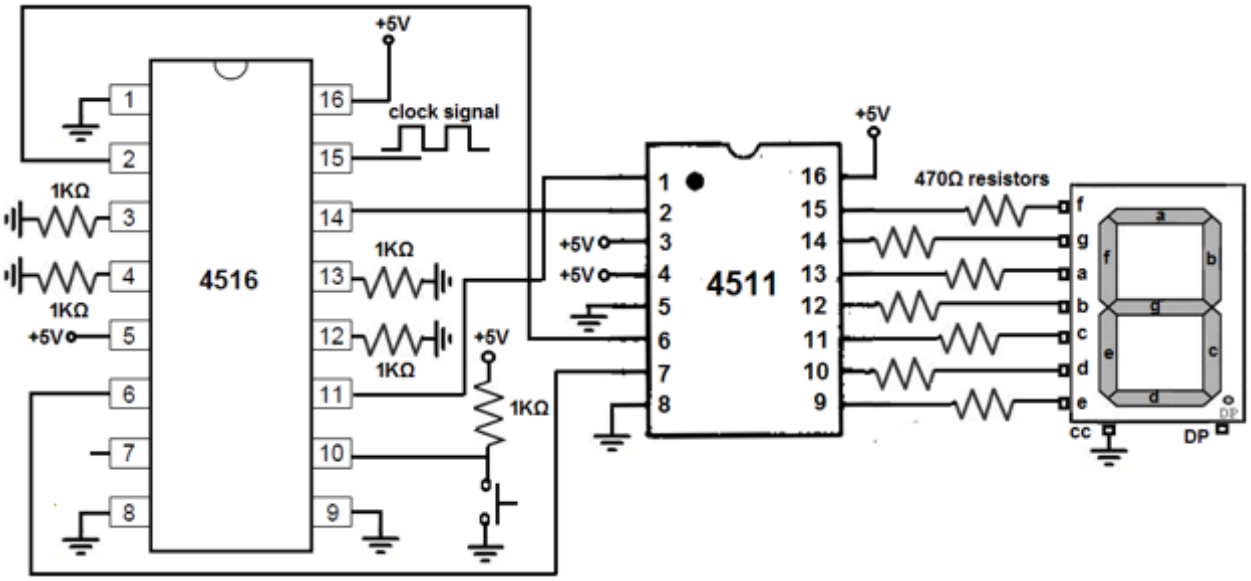
The final circuit of this marathon is the seven-segment display circuit. It uses all of the previously explained circuits (except the 4017 decade counter) to make a seven-segment display count from zero to nine or nine to zero on the press of a button. This is the final step before transferring the project to a PCB (printed circuit board). Most of the parts are the same in this circuit; all that is different is the seven-segment display itself and the wiring connecting it to the rest of the circuit. One thing to note is that the seven-segment display needs at least 470 Ω resistors, which is unlike other LEDs which can be put directly into circuits because of the internal resistors of the logic gates. The way the resistance in this circuit was achieved for the seven-segment display was to use a resistor for each lead, but since it is a common cathode (CC) variant (it connects to ground through pin three and eight), the same thing could be achieved with a resistor on either of the CC leads.

**Parts Table**

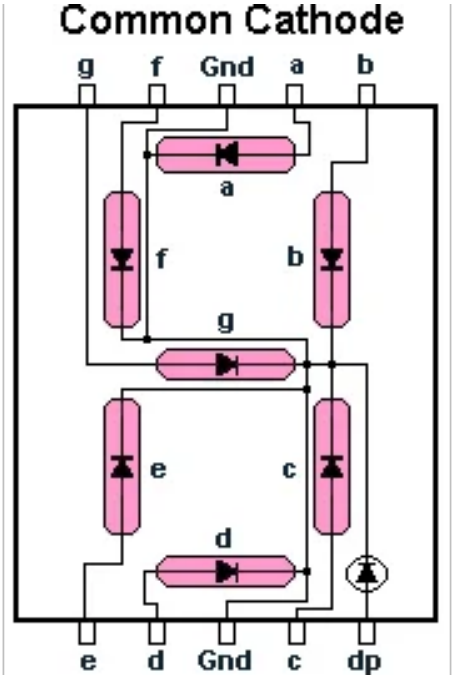
| Description | Quantity |
|---|---|
| 9 V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| Decade Counter CMOS 4017 IC | 1 |
| Up/Down Counter CMOS 4510 IC | 1 |
| BCD to Seven Segment Decoder CMOS 4511 IC | 1 |
| 1N4148 Diode | 1 |
| 10 kΩ Fixed Resistor | 2 |
| 1 MΩ Fixed Resistor | 1 |
| 470 Ω Fixed Resistor | 8 |
| Capacitor (0.1-0.01 µF) | 1 |
| Capacitor (10 - 1000 µF) | 1 |
| Button (PBNO) | 1 |
| Aces DC Breakout Board | 1 |
| Assorted Jumper Wires | 44 |
| Seven-Segment Display | 1 |

To the right is a simplified "schematic" of the circuit. Instead of showing logic gates or precise wiring, it shows the sections that have already been discussed. When the button is pressed, RC1 in the NAND gate oscillator starts its countdown and starts the RC2 pair. The square wave that the circuit uses is now being generated. This goes into the 4017 decade counter through pin 15 (clock input) and comes out pin 12 (divide by 10). This divides the frequency of the square wave by ten (which could also be achieved by adjusting the RC pairs). The 4510 BCD counter increases its count by one on every rising edge of the 4017's output. This is where the count happens. The 4511 decoder takes the BCD number and converts it to a value that is useable to the seven-segment display. This is where the new part starts.

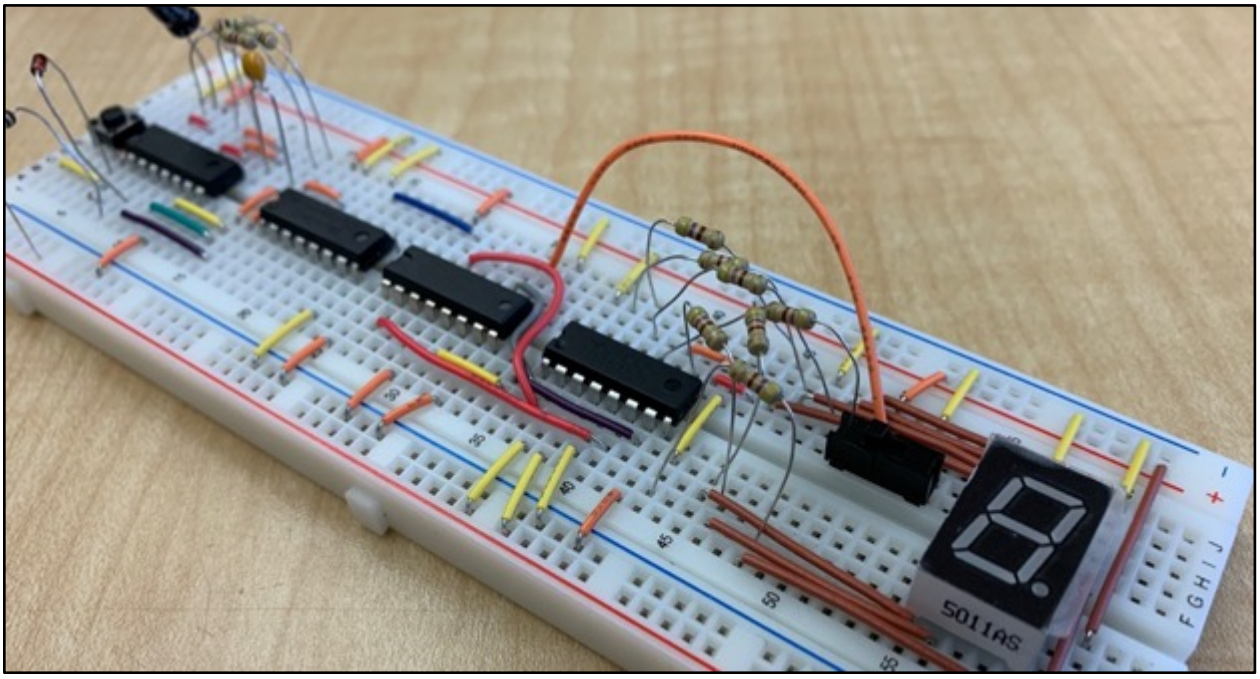




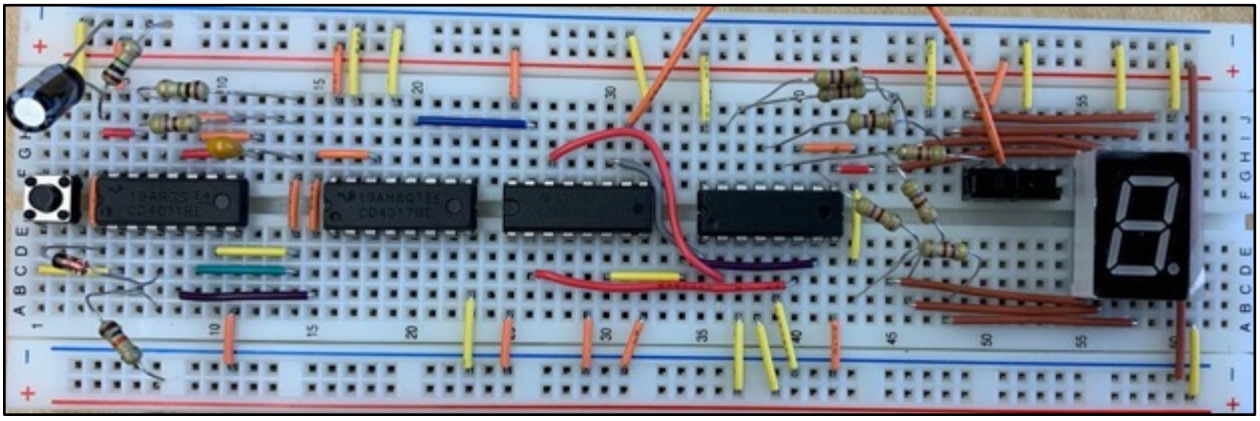

Above is a schematic of the later part of the circuit. As mentioned in the decimal counting binary section, the 4516 and 4510 are almost interchangeable; there will just be a delay after the count hits nine with the 4016. In the schematic, either would be used but the circuit uses a 4510 in real life so there is no time when the display is blank. The schematic above starts with the BCD counting chip. It takes in the clock signal and outputs its count to the 4511, the decoder. Its seven segments are each connected to one LED on the display (not including the decimal LED) through 470 Ω resistors. The CC pin is tied to ground to complete the circuit. To the right is a pinout/schematic of the seven-segment display. The pins are in no particular order, which makes the wiring messy when building. This is the only chip in the whole circuit where no logic is used, just the configuration of LEDs. Theoretically, a seven-segment display could be made on a breadboard and function the same with no logic.



## Media




A Counting Circuit Prototype Side View



A Counting Circuit Prototype Bird's Eye View

## Reflection

I did not really like this report because I felt that I understood what was going on but my logic chips were not functioning correctly. My NAND gate completely stopped working the next day, so I guess that there were already some problems with it while prototyping which probably was what was causing an inconsistent square wave and inconsistent counting speed. I also started to run out of orange wires which made it a lot more difficult to build because they are arguably the most useful wire. A couple of reflections ago I mentioned that I was a bit behind but I have caught up now and am not feeling too pressured for time yet.

## G. A Counting Circuit PCB

### Purpose

The counting circuit PCB is the start of the process of setting the counting circuit in stone. It moves the circuit from a breadboard to a circuit board where it will stay for many years.

### References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB

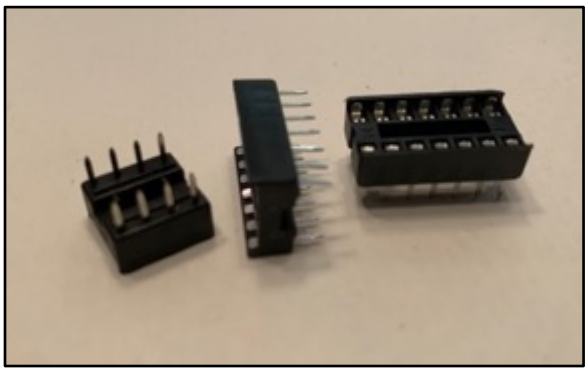
### Procedure

Moving off of breadboards and familiar ground, there are of course some new concepts that have not been touched on yet. A printed circuit board, or PCB, is a more permanent way to build a circuit. It has no loose parts; all the connections are inside of the board and the only vulnerable places are small exposed wires and pins from components soldered to the board. In this step, the circuit that has been progressing since the button input will be transferred to a PCB. The PCB comes with some new things as well, such as a way to turn off the whole circuit, soldering, chip test seats, and the board its self. The easiest to tackle is soldering. Soldering is a quick and easy way to attach electronic components. When soldering, the person soldering uses a solder pen (a device that can heat to very high temperatures) to melt solder (a mix of tin, lead, and flux) onto two things that they wish to join. Once the metal hardens again, the two items are now joined with a small piece of metal that is conductive.

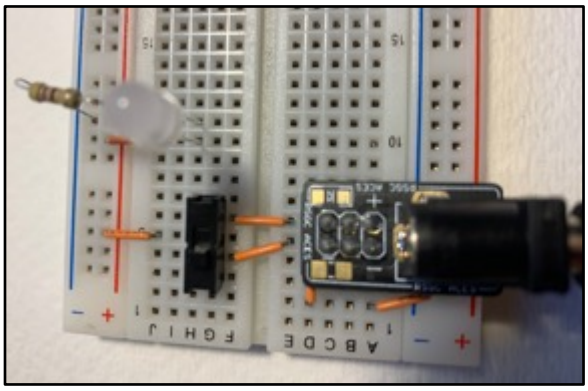
**Parts Table**

| Description | Quantity |
|---|---|
| 9 V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| Decade Counter CMOS 4017 IC | 1 |
| Up/Down Counter CMOS 4510 IC | 1 |
| BCD to Seven Segment Decoder CMOS 4511 IC | 1 |
| Button (PBNO) | 1 |
| Assorted Capacitors | 2 |
| Slide Switches | 2 |
| 470 Ω ISO Resistor Banks | 2 |
| 1N4148 Diode | 1 |
| IC Test Seat | 4 |
| Seven-Segment Display | 1 |
| 1 MΩ Fixed Resistor | 2 |
| 10 kΩ Fixed Resistor | 2 |
| 470 Ω Fixed Resistor | 1 |
| Counting Circuit PCB | 1 |

Chip test seats are used because the logic chips used in the counting circuit are sensitive to the intense heat of the solder pen, and if exposed for too long they will stop working. This is where the chip seat comes in. It is made of metal and some plastic, designed in a way that it is not damaged by the heat. Instead of soldering in the logic chips, the chip test seat is soldered in and logic chips slip right into the test seat and do not get damaged by the heat. The conductive metal in the chip seat transfers current and ground between the PCB and chip.

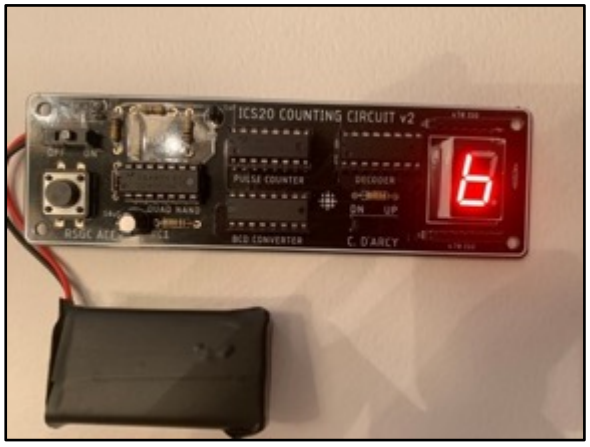


PCBs, or printed circuit boards, are the green circuit boards found in almost all electronics. However, they do not have to be green. The PCB in the counting circuit is black. The colour seen on circuit boards is the colour of the solder mask, a layer that protects the board from shorts, which is chosen by the maker. A PCB is made for only one specific circuit. There are holes ready for components to be soldered into. Unlike breadboards, there are no jumper wires involved. PCBs have all their wires inside the main insulator material of the board. All that is needed to assemble them is solder on the parts and connect the circuit to power and ground.



Now, after knowing a bit about PCBs, it is easier to explain how the on/off function works in the PCB version of this circuit. It is a feature that was never implemented on the breadboard version, but to the right is a diagram that shows how it would have been done. The power let into the circuit has to go through a switch which is either connected to ground or power. When the switch is off, there is no power available for the rest of the circuit and all leads are tied to ground. This is different than previous applications of slide switches. Previously, they have been used to control an individual component or pin of a logic gate, but here the switch is being used as a master switch that controls all power on the breadboard.

## Media




PCB Circuit Working

View of Solder Joints

## Reflection

The following reflection was written after soldering my components on the wrong side of a PCB, seemingly barring me from the rest of the project due to my oversight. Luckily, Mr. Darcy was generous enough to give me a second attempt the next day, but not without a reflective comment. He said that I would remember this small mistake for the rest of my engineering career. This really got me thinking, not about soldering a PCB backwards but about what caused it. It did not take me long to find the answer, and this is what I wrote.

December 1st:

"I am writing this reflection on December the first, the same day that I am writing about the 4511 chip, but I wanted to finish this reflection while events were still fresh in my mind and because I think that this reflection is potentially the most important lesson in my DER so far.

I have known for as long as I can remember that I suffer from taking things too quickly and making decisions without fully considering every part of the situation. If you were to ask me what my biggest academic weakness is, my answer would be that I am unable to stop and think before I decide that something must be the answer, and I rarely give it a second thought. I think that I continuously make this same mistake over and over again for two reasons. They are both gifts and curses. Firstly, for a long time, I was able to spend five seconds on almost most questions in any class and come up with the correct answer. It is only recently that I have realised that this is not the case anymore, and as school gets more difficult more thought is needed. I guess I have not been able to get out of my old mindset and see the true reality of my situation which is that I have to change my approach if I want to keep being right consistently. The second reason is that I, until now, have not had a reason to change anything. Sure, overlooking something on a question because I did not take enough time to look everything over can be annoying, but it has never had consequences before. On homework or classwork, wrong answers are simply opportunities to learn from your mistake and nothing more. On a test, I will probably catch something that I overlooked by the second or third check of my work. But today, when I soldered my PCB backwards, essentially closing the doors to the final two parts of this report, I finally got consequenced for my bad habit. I am honestly kind of relieved, because in hindsight, I was definitely going to face the consequences of overlooking something eventually, and there are much worse things that I could have messed up on. But at the end of the day, hindsight is hindsight. Everything is so much more obvious when looking at it in the past. What happened today allowed me the opportunity to really reflect on myself and gave me the chance to make a change where I can hopefully start taking things more slowly and always try to pay attention to every detail."

While it is cheesy, I was completely serious about what I said. Even though I got a second chance, I have been trying to slow things down and I have already been able to catch multiple errors on the first try that I otherwise would have missed.

## H. A Counting Circuit PCB Case

### Purpose

The PCB case is the final step of the counting circuit project. It hides the battery and solder joints and ties a nice bow around the presentation of the final circuit on the PCB.

### References

Project Description: http://darcy.rsgc.on.ca/ACES/TEL3M/2122/TasksFall.html#CCB

### Procedure

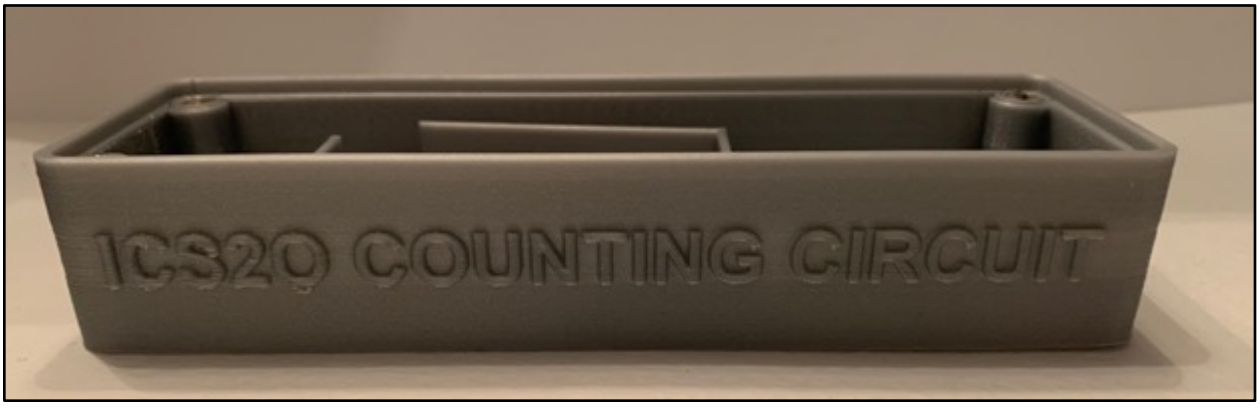
The PCB case is pretty self-explanatory. It is a case specially 3D printed for the counting circuit. It houses the battery and the PCB mounts nicely on top of it with four plastic screws. Something to note is that the metal casing of the battery can interfere with the solder joints on the circuit, so it is important to cover the part of the battery that is exposed to the joints with an insulator such as electrical tape. Electrical tape is made out of a type of PVC plastic and is non-corrosive. This is useful since it is being used to wrap a battery which will corrode over time.

One thing worth mentioning is that the screws strip easily if tightened too much and they do not come out well once stripped since methods of removing them such as pliers squeeze the soft plastic and completely deform it.

| **Parts Table** | |
|---|---|
| Description | Quantity |
| 9V Battery | 1 |
| NAND CMOS 4011 IC | 1 |
| Decade Counter CMOS 4017 IC | 1 |
| Up/Down Counter CMOS 4510 IC | 1 |
| BCD to Seven Segment Decoder CMOS 4511 IC | 1 |
| Button (PBNO) | 1 |
| Assorted Capacitors | 2 |
| Slide Switches | 2 |
| 470 Ω ISO Resistor Banks | 2 |
| 1N4148 Diode | 1 |
| IC Test Seat | 4 |
| Seven-Segment Display | 1 |
| 1M Ω Fixed Resistor | 2 |
| 10k Ω Fixed Resistor | 2 |
| 470 Ω Fixed Resistor | 1 |
| Counting Circuit PCB | 1 |
| Counting Circuit PCB Case | 1 |

### Media

Project Video: https://youtu.be/2ckSo_JbLZI




PCB Case


Final Product

## Reflection

I am going to make the reflection on the PCB case pretty short as it was pretty uneventful (besides stripping two screws while unscrewing the PCB for pictures). I guess since this is the final reflection for almost a year, I should probably reflect on my whole two-month experience in hardware.

While I came into this class already knowing some things about hardware and logic, I did not fully appreciate the depth behind the knowledge. I could tell you the inputs and outputs of an AND gate the day that I walked in but I could not have told you which lead of an LED was which, or even which way current flows. This course has given me an astoundingly thorough knowledge base of both simple and complex theory and application of hardware concepts that I did not even know that I was lacking on my first day. This short 35-day course has already been one of the most interesting experiences I have ever had. I mean, a course about everything that I have ever been curious about related to electronics? It sounded too good to be true, like there would be a catch, except there was no catch. I have never heard anyone else from other schools talk about a class where they get to learn about genuinely interesting subjects and then apply their learning directly after. I think that approach of applying learning is infinitely more engaging and motivating, at least it was for me.

On every day one this year, I have looked forward to hardware. It is a highlight of the day where I get to apply myself and I know that I am going to miss it for the remaining half of the day ones left. Of course, I would like to include a thank you to Mr. D'Arcy who taught us life skills, lessons, and of course, lots about hardware.

# ICS3U

# Project 2.1 The 555 Time Machine

## Purpose

The 555 Time Machine circuit shows the inner workings of the 555 integrated circuit (IC) and introduces new concepts such as latches and comparators. The applications of the square wave output from the 555 are numerous and understanding the 555 and its output opens many new doors for creations in the future.

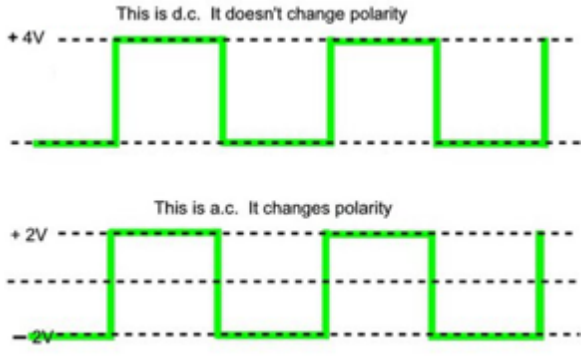
## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#555
NE555 Data Sheet: http://darcy.rsgc.on.ca/ACES/Datasheets/NE555N.pdf

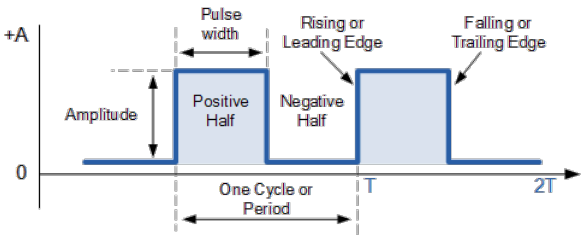
## Procedure

### The Square Wave

The purpose of the 555 is to generate configurable square waves. While the concept of a square wave is not a new topic, this project warrants a deeper look into its workings and application. A square wave is a clock pulse whose output alternates between 0 and 1. Square waves have many applications in the real world that might not be obvious. For example, the diagram on the right shows DC and AC square waves. AC power comes in a square wave, except instead of alternating between 1 and 0, the current switches directions, which is illustrated with negative voltage.



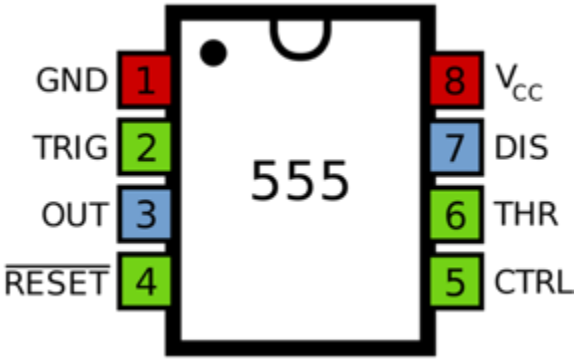

To understand how a square wave can be configurable later, it is important to know the associated terminology. First are the rising and falling edges. What differentiates square waves from other waveforms are the edges. In a square wave, the rising and falling edges take up negligible time. These edges are the time that it takes the output to change from 0 to 1 and 1 to 0. Essentially, in a square wave, the output is only ever 1 or 0 because of the extremely short rising and falling edges, during which the voltage output is somewhere in between source voltage and ground. This differs from a wave with a longer rising and falling edge which would output values between 0 and 1 during the rise and fall. Next, the amplitude is simply the height of the wave, which represents the voltage output. Finally, there is a period. A period is the combined time of one positive half and one negative half. When the output is 1 it is a positive half and vice versa with negative halves. The ratio of the positive half to one period is called the duty cycle.




As mentioned before, square waves have some important applications. Every computer on earth utilizes square waves. Inside the processor of computers, components need time after each operation to settle into their new states. An operation is simply a change from 0 to 1 or 1 to 0. If another operation is attempted before the components' states completely solidify as a 0 or 1, the outcome will be wrong. To solve this problem, processors use a square wave. One time every pulse, generally on the rising edge, the processor will start to complete operations. No new operations will be started until the next clock pulse, giving time for signals to settle. Granted, these clocks in computers operate in billions of hertz a second instead of a few dozen, but the idea is similar.

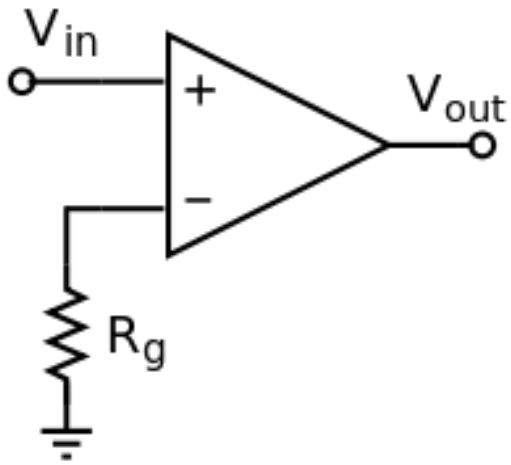
## The 555 IC

On the surface, the 555 looks like quite a simple chip. For example, ground, supply, reset and output pins are familiar already. Using the actual 555 to generate a square wave is not too hard a task. The difficulty comes when the 555 is opened up and the inner workings present themselves. Before looking at how the 555 works under the hood, some new ideas need to be discussed. Firstly, there are comparators. Comparators essentially take two analog inputs; if input a is greater than input b, the output turns high. Secondly, there is the SR latch. Latches are very basic memory devices and are capable of storing one bit of information.




## Operational Amplifiers

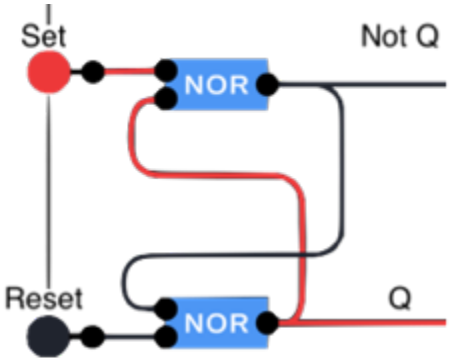
As previously mentioned, the 555 makes use of comparators. In the mid-level 555 circuit, the comparator used is the LM358 operational amplifier (op amp). Note that op amps are analog components, so their inputs and outputs are not always 0 or 1. The op amp can be wired in a few different ways but to use it as a comparator it must be wired in an open loop, meaning that there is no feedback taken from the output. The open loop amplifier functions as a comparator by amplifying the difference between the two inputs. The op amp's output can be calculated with the equation $V_{out} = A_{OL} (V_+ - V_-)$. $A_{OL}$ is equal to the open loop gain of the op amp. The open loop gain is simply the name assigned to the number by which the difference in outputs will be multiplied. An op amp with an open loop gain of 100 would output the difference in inputs multiplied by 100.

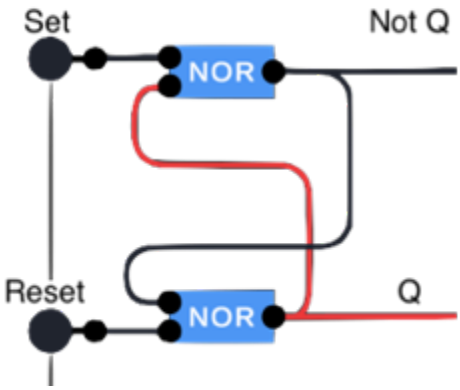



## The SR Latch

The SR latch stands for set-reset latch. A latch is a device that is capable of storing one bit of information. In this build the SR latch used is a NOR latch, meaning that it uses NOR gates. Latches are closely related to flip-flops since they both store one bit of information and are built similarly. The difference is that latches act immediately on inputs, but flip-flops are edge reliant as discussed earlier and have to wait for the next clock signal to change state. In bistable mode, the 555 can be used to make a flip-flop circuit.

The SR NOR latch is depicted on the right. As seen in the diagram, the Set input, when high, will turn on the Q output. However, when the Set input then turns low, Q stays on. No matter how many times the Set input is now turned from high to low, Q will stay high and NOT Q will stay low. However, if Reset turns high, Q will turn low and NOT Q will turn high. Now Reset can change from high to low and nothing will happen. If Set is pressed now, it will revert the latch to its original state. The application of this is that the latch has the ability to "memorize" its state. Latches are examples of sequential logic. The output relies not only on the present inputs but also on past inputs. Because of this function of sequential logic, things such as truth tables, which are helpful in understanding the function of combinational logic, prove useless as the output is independent of the current input. Interestingly enough, there are many truth tables for the SR NOR latch on the internet.



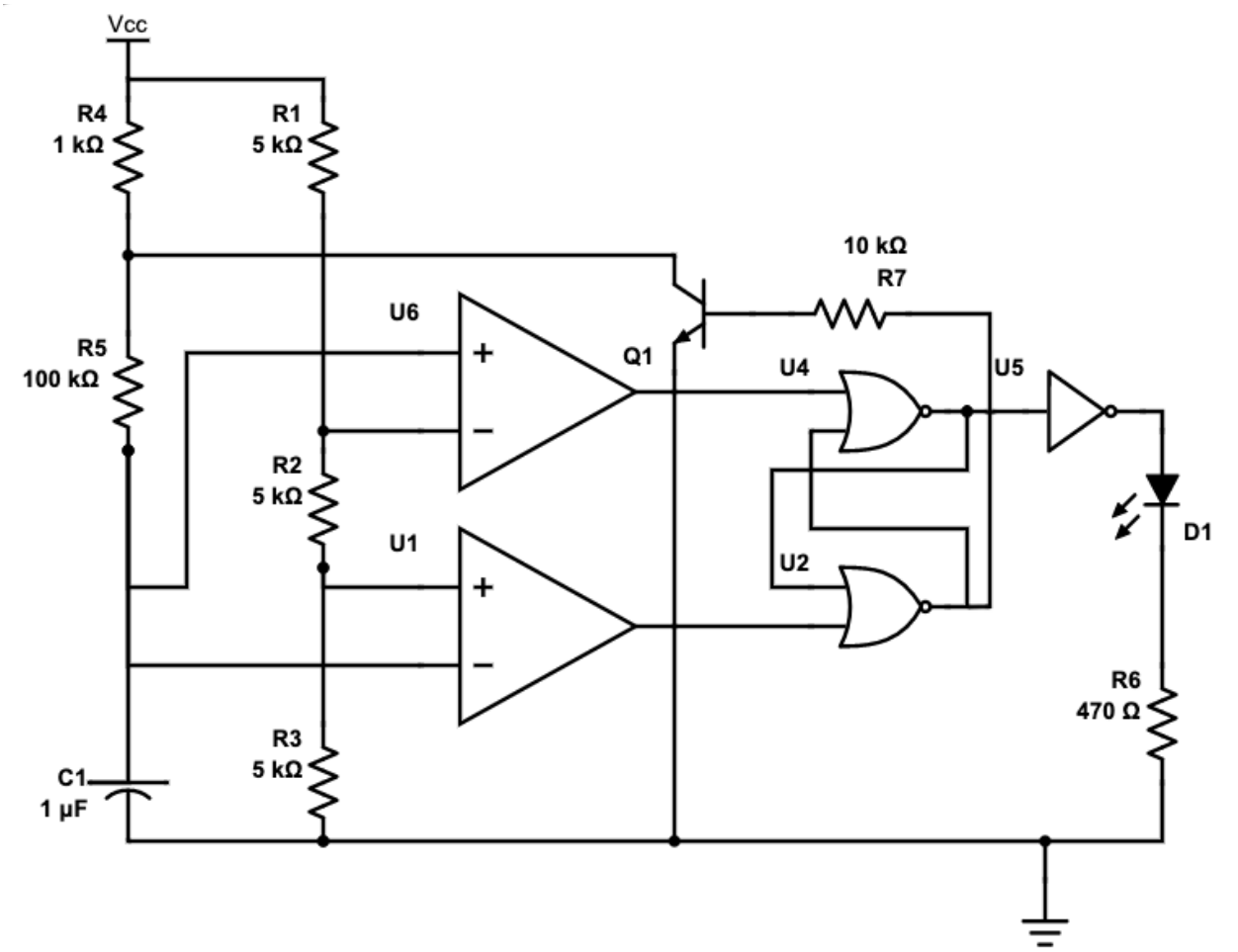



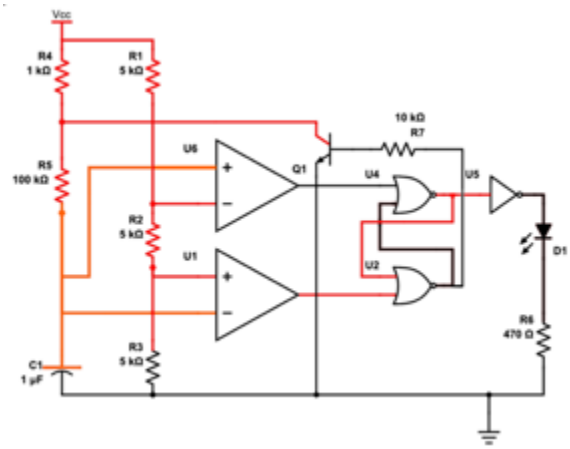

In the 555 mid-level circuit, the inputs to the SR latch come from the outputs of two comparators. Together these parts are the logic of the 555. After understanding the components that make up the timing circuit, it is now time to take a look at how they come together to create a clock pulse.

## The Mid-Level 555 Timing Circuit

The mid-level 555 circuit takes a look at the internal functioning of the 555. It replicates the function of the 555 by using the same parts that are normally hidden inside the chip. As discussed, this includes two open loop op amps acting as comparators and an SR NOR latch as well as some other components. One such component is a NOT gate. As previously mentioned, op amps are analog devices and their outputs are not always exactly a 0 or a 1. In this circuit, op amps are mixed with digital components, and the NOT gate is there to rectify their weak signals. Without the NOT gate the circuit could give off inconsistent results. That is the only function of the NOT gate in the circuit so it will be largely ignored while analyzing the function of the 555. Below is the schematic of the mid-level circuit.

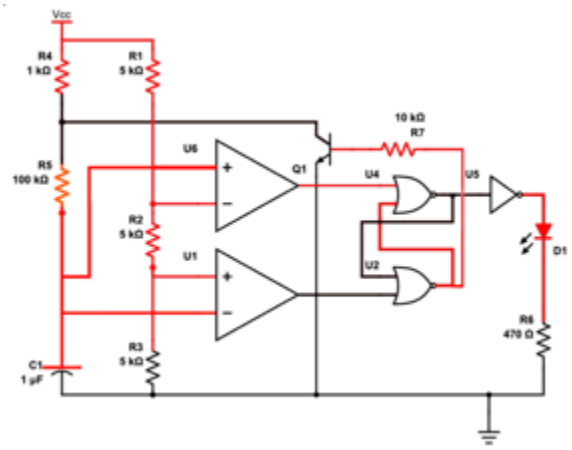
| **Parts Table** | |
|---|---|
| Description | Quantity |
| LM358 Operational Amplifier IC | 1 |
| NOR 4001 CMOS IC | 1 |
| NOT 4049 CMOS IC | 1 |
| NPN BJT 3904 Transistor | 1 |
| 10 µF Capacitor | 1 |
| 1 kΩ Fixed Resistor | 1 |
| 4.7 kΩ Fixed Resistor | 3 |
| 10 kΩ Fixed Resistor | 1 |
| 100 kΩ Fixed Resistor | 1 |
| Green LED | 1 |
| Assorted Jumper Wires | 24 |



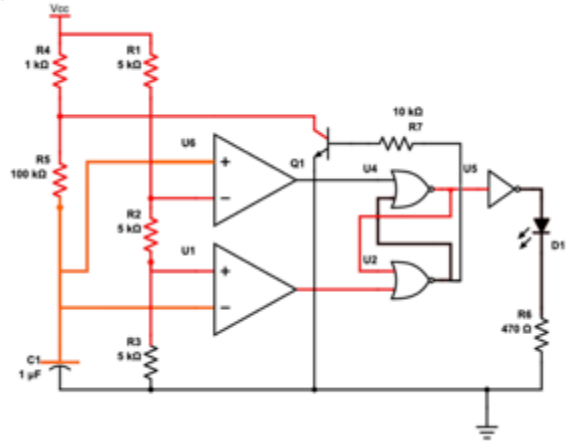



Flipping the power on, nothing happens. The LED stays off. Inside the circuit, much more is going on. Firstly, the capacitor that was once empty is starting to charge (illustrated by orange wires). The voltage of these orange wires is climbing, but at the moment it is very small. The three 4.7 kΩ resistors in series divide the source voltage and present 3.33V to the inverting input of the first (top) comparator and 1.67V to the non-inverting input of the second (bottom). For now, the inverting input of the second comparator is less than 1.67V and therefore the comparator is outputting a 1. This goes to the NOR gate in the latch and locks it (meaning the NOR gate cannot output a 1 while the comparator is on regardless of the other input). Meanwhile, the other comparator has a non-inverting input that is smaller than the inverting input, so the comparator does not output a high. With neither of its inputs a 1, the top NOR gate outputs high and latches the other gate. The climbing voltage in the orange wires will eventually surpass 1.67V (turning the second comparator's output low) but it will not matter because the bottom NOR gate is latched by the top gate.

The capacitor has now fully charged. The 5V present at the non-inverting input of the first comparator is more than the 3.33V at the inverting input, so the comparator is now outputting a 1, locking the top NOR gate. The other comparator is now outputting a low as expected, and since the other input to the bottom NOR gate is a 0 as well, the bottom NOR gate outputs a 1, latching the top gate. However, this outputs a 1 to the base pin of the transistor, letting the comparator discharge through the 100 kΩ resistor. This means that the voltage of the threshold and trigger (the comparator inputs that rely on capacitor charge/ discharge) input's voltage is now decreasing.




Back to square one. The image to the right is the same as the first diagram of this series. The voltage in the trigger and threshold wires decreased below 3.33V (turning the top comparator output low) and then below 1.67V (turning the second comparator output high). This means that the bottom NOR gate had a high input. NOR logic then tells us that the bottom gate started to output low. With both of its inputs low, the top NOR gate is then turned on. With the bottom NOR gate off, the transistor stopped allowing the capacitor to discharge and once again the capacitor is charging and the trigger and threshold voltage is climbing.




## Formulas and Configuration

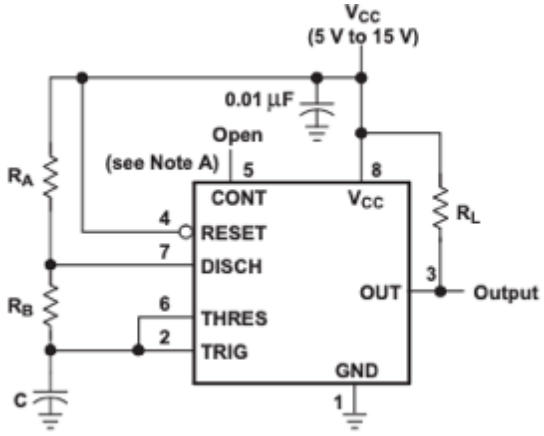
Many times, it has been mentioned that the 555's pulse can be modified. This is done by changing the values of external components. There are formulas that predict the effects of changing certain values of resistors and capacitors.

To the right is a schematic of an astable 555 wired quite typically (RL is not relevant here). Resistors and capacitors act as variables; these formulas help to choose what value to use based on a desired output.

$$\text{Duty Cycle} = \frac{R_B}{R_A+2R_B}$$

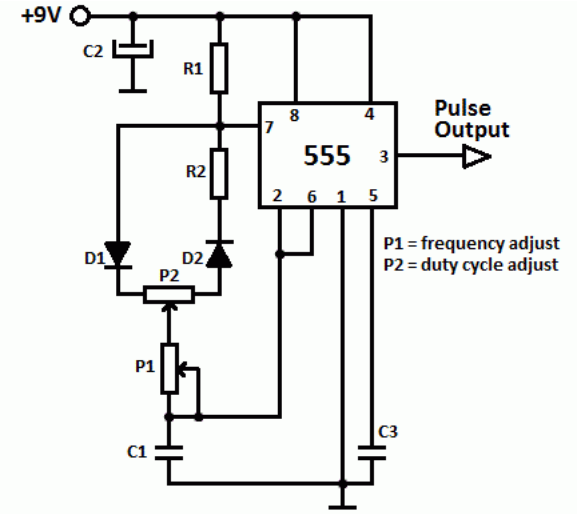
$$\text{Frequency} \approx \frac{1.44}{(R_A+2R_B)C}$$

$$\text{High to Low Ratio} = \frac{R_A+R_B}{R_B}$$



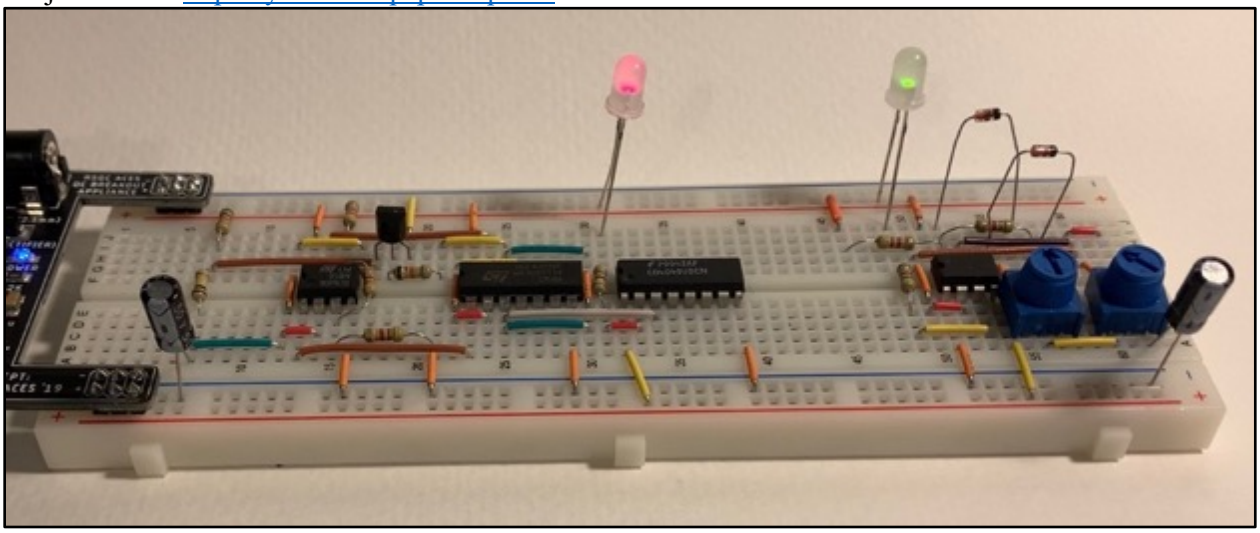

Another thing that one may wish to introduce to the circuit are variable resistors. These add the ability to change the frequency and the duty cycle on the fly. It is simple enough to add a variable resistor to change the duty cycle or the frequency, but there is an added challenge to controlling both. Turning a potentiometer will affect both the total resistance (frequency) and the ratio between the resistors (duty cycle).

To the right is a circuit that chancges the frequency and duty cycle independently with two potentiometers. It looks scary with the added diodes, but their function is just to make the current flow through both the A and B lead of the potentiometer in its total journey. All this does is make sure the total resistance experienced because of the potentiometer is always the same no matter the orientation of the "dial". The two routes added together will always have the same resistance meaning that the frequency will stay the same when P2 is turned. The changes to P1 change the potentiometer from a variable resistor to a voltage divider, meaning that the total resistance of the potentiometer will never change. That means the ratio of P1:P2 cannot be affected by turning P1.




## Media

Project Video: https://youtu.be/3pcp7TdqDEY



Both Circuits on One Breadboard

## Reflection

The first of many DERs this year. I think that it went pretty well. The hard circuit certainly helped to shake off the rust quickly after almost nine months without hardware. The biggest problem that I encountered was that I was too busy to get any work done on my DER before late Thursday night. I was not used to a full schedule after the summer and am still re-learning how to be productive. I kept pushing off starting the DER because I had to complete the homework due the next day. My next biggest problem, which was not a great combo with not having much time, was that I found too many things to write about. I have heard from other students that their DER is once again far shorter and very likely more concise than mine. One of many examples of finding too much to write about was when I learned that the duty cycle of the 555 was customizable with a potentiometer. I immediately started researching how to build such a circuit and 30 minutes later I found myself funnily enough on the same website that you shared with us the next day. I then spent another 30 minutes understanding how it worked, and then another 10 to build followed by 20 to write about it, meaning I spent 1:30 on a half-page add-on to my already lengthy DER.

# Project 2.2 74HC595 Shift Registers

## Purpose

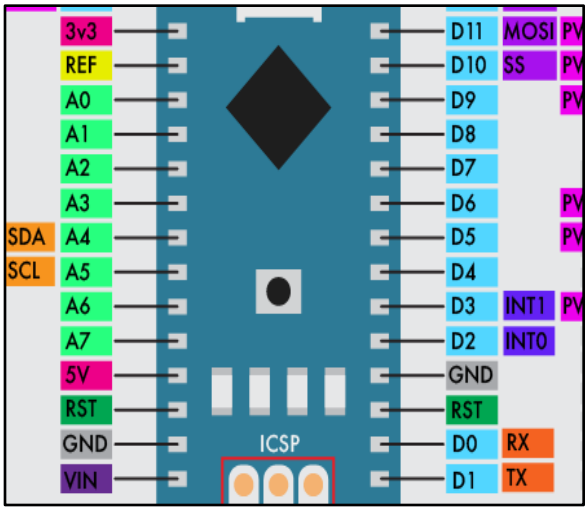
The purpose of project 2.2 is to learn about the workings and applications of shift registers, and how to use and integrate them into circuits. These points culminate in the creation of a replacement for the Nano Coding Companion (NCC).

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#ShiftRegister
D Flip-Flop by Ben Eater: https://www.youtube.com/watch?v=YW-_GkUguMM&ab_channel=BenEater

## Procedure

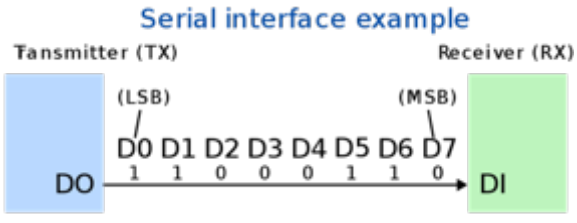
### Microcontrollers



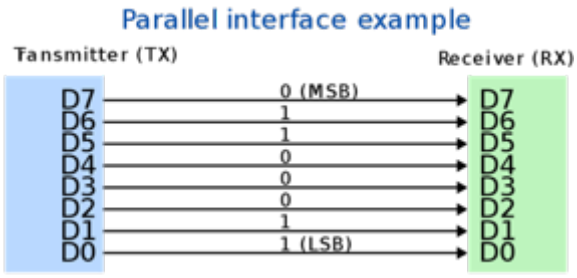

Project 2.2 marks the first appearance of microcontrollers. Microcontrollers allow for the addition of software to physical circuits. The possibilities that microcontrollers open up are almost endless, and this project captures that with the freedom it allows for creativity. In this project, the microcontroller of choice is the Arduino Nano, or more accurately, the ATmega328p. The ATmega328p is a microcontroller chip, and the Arduino Nano is an interface that allows the 328p to be easily integrated with breadboard creations. Seen to the right is the pinout of the Nano. It comes with 14 digital-only pins and eight analog/digital pins (though not all analog pins can be addressed). The Nano interacts with the circuit through these pins. It is capable of presenting either a digital 1 or 0 to any of these output pins, and can also read the binary value present on each pin. In addition, the analog pins are capable of reading voltage values between a digital 1 and 0. As well as being able to read analog values, pins 3, 5, 6, 9, 10, and 11 are capable of presenting a pseudo-analog output to a circuit. The Nano does this with pulse-width modulation, or in other words by presenting an average output of the desired analog voltage value through varying duty cycle.

### Shift Registers

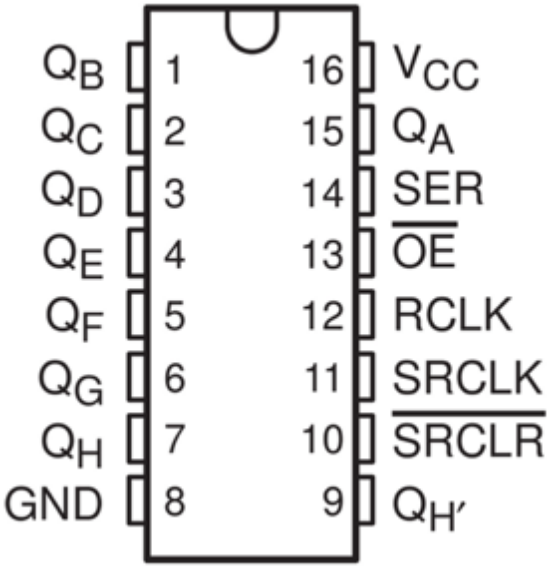
Shift Registers are a set of powerful chips capable of greatly increasing the capability of builds. However, there are a handful of different types of shift registers. In this project, the focus is on the 74HC595 serial in parallel out shift register. As its name suggests, the 74HC595 takes a serial input and outputs a parallel output. Serial communication is a new concept, and to quickly summarize, serial communication is the transmission of data through a sequence of 0s and 1s on a single communication line. To communicate serially, both the receiver and transmitter have to agree on the rate of communication, called the baud rate.



This differs from parallel communication where every bit has its own communication line. What the 74HC595 shift register does is take a serial input of 0s and 1s in sequence, and converts them to a parallel output of the same value. This has the effect of extending the number of output pins available on the Arduino. As well as serial in parallel out, there are also parallel in serial out, serial in serial out, and parallel in parallel out shift registers.

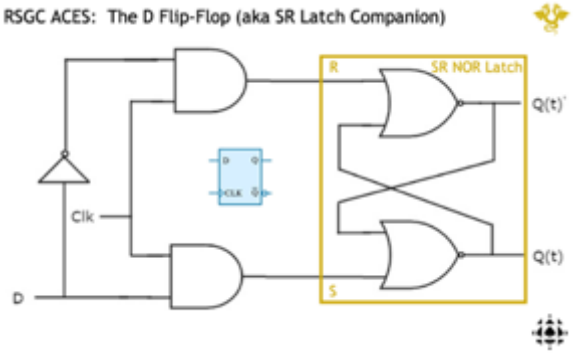


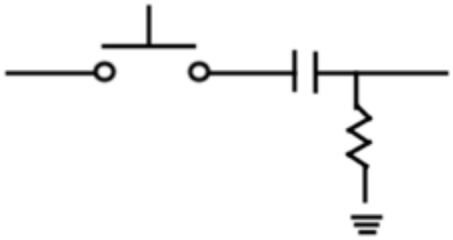

To the right is the pinout of a serial in parallel out (SIPO) shift register. Each shift register has eight outputs, $Q_A$ through $Q_H$. This means that individually, SIPO shift registers can output eight bits of serial data at a time as their parallel equivalent. To run a shift register, only three pins have to be connected to an Arduino or some other input. Those pins are pin 14, pin 12, and pin 11. Pin 14 is the data or serial pin. This is where the serial data is presented. Next, pin 11 is the clock pin. When the clock pin receives a digital 1 input, the register saves the current input to the data/serial pin as the output for $Q_A$. The previous output that was saved for $Q_A$ is shifted (hence the name shift register) to the output for $Q_B$. $Q_B$ is also shifted to $Q_C$, $Q_C$ to $Q_D$, and so on. $Q_H$, being the final output, will not be able to shift its saved output. It is important to note that the outputs do not change, only their saved value. Finally, pin 12 is the latch pin. To change the outputs, the latch pin has to be toggled on and off. The user can make as many changes to the saved value for each pin as they desire, but the outputs will only output that change once the latch pin receives a high signal. When the latch pin receives a high signal, it will update all outputs at once.



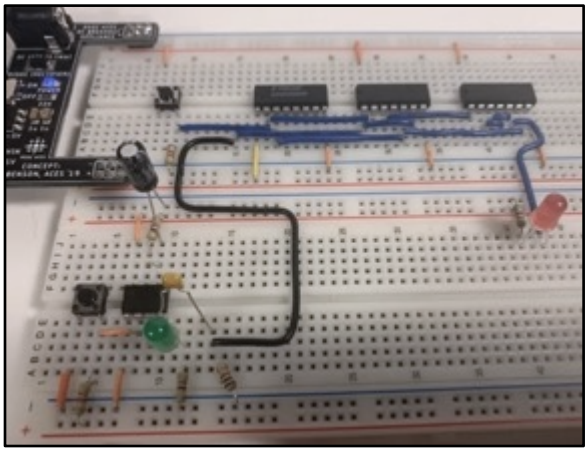

## Inside a Shift Register

To the right is a schematic of a D flip-flop. The internals of a shift register are basically just eight D flip-flops where input D is the output of the last flip-flop. Functionally, D flip-flops store D as the output on every rising edge of the clock input. Any changes to D in-between clock cycles will have no effect. To do this, a D flip-flop first takes D and the clock and compares them. If they are both high, then SET is high on the internal SR NOR latch and the output goes high. If D is low, it is turned high by the NOT gate and is then compared with the clock. If they are both (NOT D and clock) high, the output then goes low. If in either scenario the clock was low, nothing in the output would change. This means that the circuit mimics what D was at the most recent clock signal.



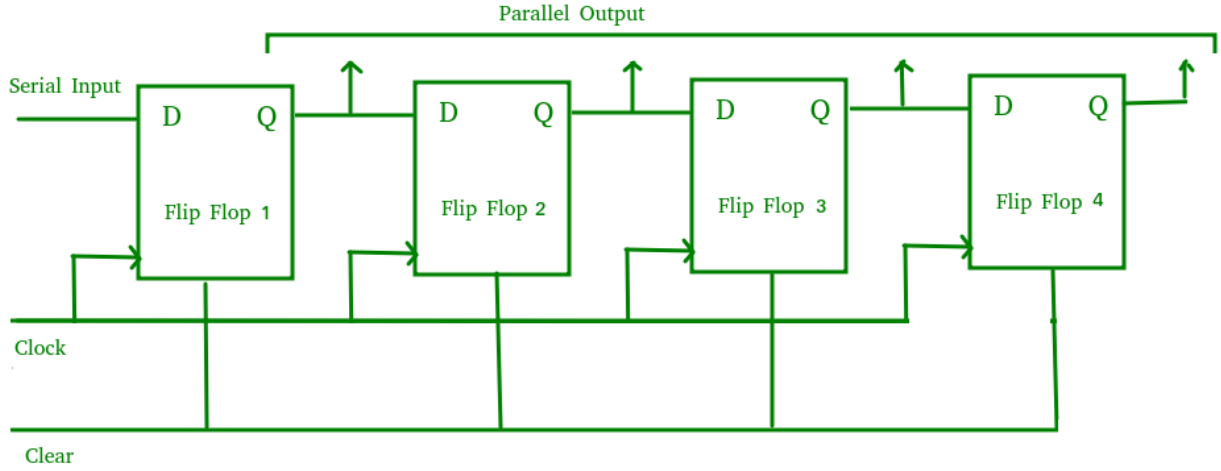


While the above paragraph is not exactly wrong, it does not explain the full picture. With the above schematic, the output would be the same as the D input at any point of the clock cycle as long as it was positive. This is actually how D flip-flops work, but first the clock input has to be translated into only its rising edges so that outputs can only change on a new clock pulse. There are many ways to do this, but one of the simpler ones uses a resistor and a capacitor. As shown to the right, this simple circuit can turn a constant input into a single quick pulse. Current is only present on the right side after a button press for the short time the capacitor takes to charge. The pulse length can be calculated with the formula $t = R_1 \times C_1$.



The image to the right is a D flip-flop. The blue wiring is identical to the first schematic of the flip-flop, but on the bottom board, there is some new wiring. A 555 is used to debounce a button input, and after that, the clock signal is turned into a pulse with a 1 kΩ resistor and a 0.1μF capacitor. When inputting these values into the formula, it turns out that the pulse length is turned into 0.1 milliseconds. While this time is not instant, it closely resembles the rising edge of the clock and indeed in the circuit to the right, the red LED only changes state on the rising edge of the clock.

The final product looks something like the image below, only with eight segments. One final quality of a shift register is its bit significance. Shift registers can be least significant bit first (LSBF), or most significant bit first (MSBF). This is pretty self-explanatory, but it dictates the way that values should be inputted into the register. Internally, this only changes the direction that the bits shift. In an LSBF register, $Q_A$ shifts towards $Q_H$. In an MSBF register, $Q_H$ shifts towards $Q_A$. The 74HC595 is an LSBF register.



### Daisy-Chaining Shift Registers

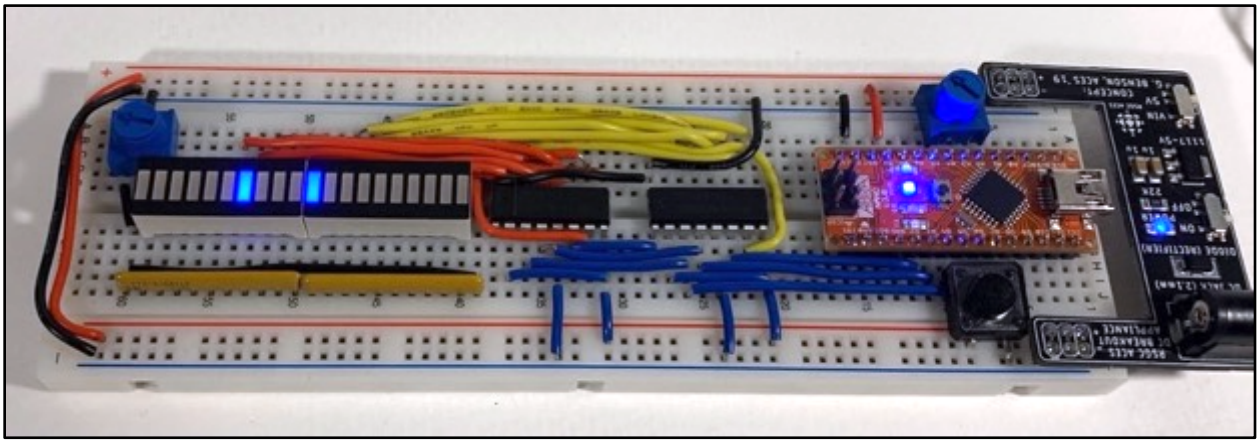
On the previous page, it was mentioned that an *individual* shift register could only have eight outputs. This wording was intentional because shift registers are stronger together. They can be daisy-chained to have practically infinite outputs. This is also relatively simple. On the earlier pinout, there is a pin named $Q_H$'. This pin is the secret to daisy-chaining multiple shift registers. To daisy-chain shift registers, the first register will have the serial input as its input into pin 14, but all subsequent registers will use the previous registers' $Q_H$' pin as their pin 14 input. If the clock and latch pins of every shift register are all attached to the same inputs, the outputs will be expanded to $8 \times n$ where $n$ is the number of registers.

While pin nine is named $Q_H$' (meaning that it should be electrically identical to $Q_H$), that is not exactly what it is. $Q_H$' represents the last bit that spilled out when the bits in the register shifted. This means that the input into pin 14 of the shift register down the line is receiving the spilled bit from the register before it as its input. This shift register now stores that bit, meaning that it has not spilled out, it is the $Q_A$ of the next register. With this, all bits are preserved until the very end of the line and the shift registers act as a single chip.

### Rules and Breakdown

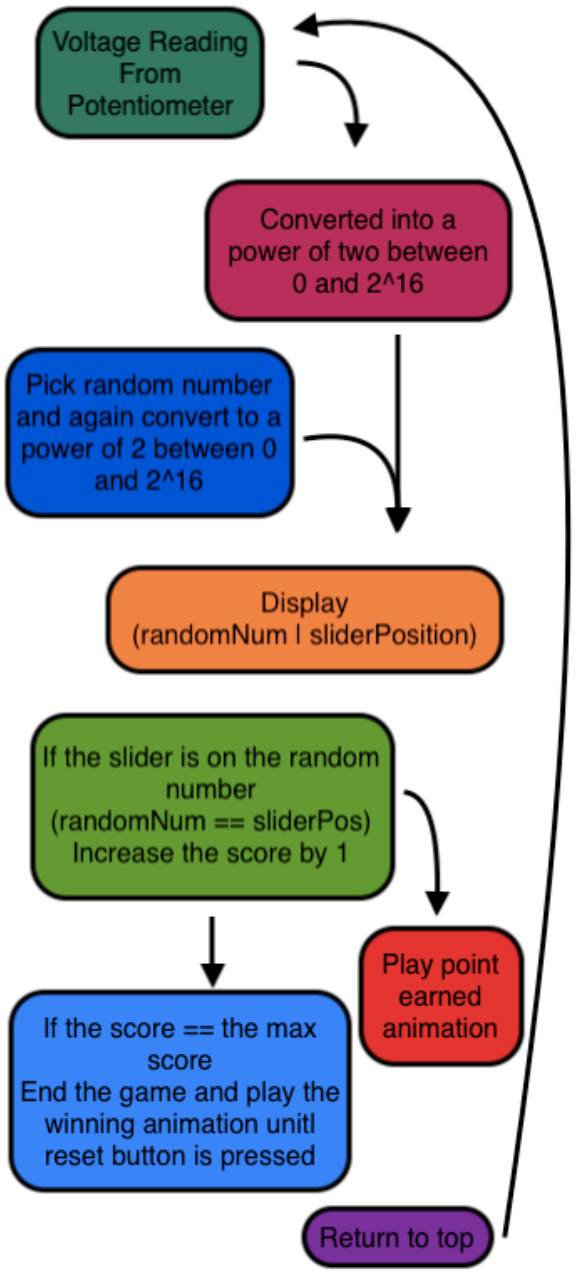
The chosen route for this project was to make a simple game. It has customizable difficulty and max score which the user can set within the code. To play the game, the user can control a "slider" (one illuminated bar on the bar graphs) by turning a potentiometer. The objective is to move the player's slider on top of another slider that is generated in a random position. The randomly generated slider moves around at a rate defined by the difficulty. If the player's slider is on the random bar as the random bar moves, the player is awarded a point and a short animation will play to let the player know that their score has increased. Once the player's score reaches the score needed to win (which can be defined in the code), the game ends and the screen begins to flash. To play again, the player can press the reset button.

| **Parts Table** | |
|---|---|
| Description | Quantity |
| Arduino (ABRA) Nano | 1 |
| Push Button (Normally Open) | 1 |
| 100 kΩ Potentiometer | 2 |
| 74HC595 Shift Register | 2 |
| 10 LED Bar Graph | 1 |
| 1 kΩ Fixed Resistor Array | 2 |
| Assorted Jumper Wires | 34 |



The Final Game

To the right is a block graphic of what is happening inside the Arduino to make the game run. To start, the Arduino reads the voltage present on the input pin from the potentiometer used to control the slider. This number then has to go through some math where it is converted from a number between 0 and 1023 to a number between 0 and 15, increases by one (it now matches an LED on the bar graphs), and finally, two is raised to the power of the number between 1 and 16. This means that the voltage reading is now a power of two between 2 and $2^{16}$. This means that it can now be masked and displayed as only one LED segment. Next, the code picks a random number and puts it through the same math process to make it an acceptable number to display (this only happens 1/50 of the time by default, the random number normally stays the same). This random number is the bar that the player has to match with their slider. The bitwise OR operator is then used to display both bars at once on the bar graphs.




Next, the program checks to see if the player's slider is on the random slider. If it is, the player earns a point and a short animation is played to let the player know that they got a point. After the animation plays the code checks if the current score is equal to the score needed to win the game, and if it is, the winning animation will play indefinitely until the reset button is pressed. If the winning score has not been reached, the loop ends and returns to the top.

Outside the Arduino very little happens, but as per the project description, it should be mentioned that there is a second potentiometer that can be used to change the brightness of the bar graphs. This is not done through the Arduino. The potentiometer acts as a variable resistor. The two resistor arrays are connected to ground through the potentiometer, so as the potentiometer changes resistance, the total resistance experienced by the bar graphs changes. This could of course be done with pulse-width modulation but when tasked to make an analog component have an analog effect on a circuit, it is better to avoid converting analog to digital and then digital to analog when possible.

## Code

```
// Project   : Shift Registers
// Purpose   : A game where the player matches a controllable slider with one that
               is randomly generated.
// Course    : ICS3U
// Author    : Liam McCartney
// Date      : 2022 10 28
// MCU       : 328p
// Status    : Working
// Reference : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#ShiftRegister

// Variables :
#define data 2
#define latch 3                       //Defining pins on the shift register
#define clock 4
#define duration 10                   //There is no practical reason to change this,
                                        but it is possible
uint8_t timing = 0;                   //Used to change the random less often than
                                        the screen refreshes
uint16_t objective;
uint8_t score = 0;
#define maxScore 5                    //Changes the points needed to win
#define animationLength 8             //Changes the length of the animation when the
                                        player earns a point
#define difficulty 50                 //Sets the "difficulty" (determines how often
                                        the random bar resets)
#define reset 7                       //Defines the reset pin

void setup() {
  Serial.begin(9600);
  pinMode(data, OUTPUT);
  pinMode(latch, OUTPUT);             //Initializing shift register pins
  pinMode(clock, OUTPUT);
  clearDisplay();                     //Clears the display from any previous
                                        programs

  pinMode(A2, OUTPUT);
  pinMode(A4, OUTPUT);
  digitalWrite(A2, 1);
  digitalWrite(A4, 0);                //Sets up the potentiometer control
  Serial.println("Hello World");
}

void loop() {
  uint16_t position = 2 << (analogRead(A3) >> 6);     /*In short, the math here
takes a value between 0 and 1023 and converts is to a power of 2 (between 2^0 and
2^16). This is needed to display only 1 bar on the bar graph.*/

    if (timing == difficulty) {       //This statement is only true
                                        1/difficulty of the time, meaning
    timing = 0;                       //that the random bar only changes every
                                        (difficulty) frames
    if (objective == position) {      //If the player's bar is on the random
       score++;                         bar, they earn a point
     for(uint8_t celReps = 0; celReps < animationLength; celReps++) {
        setDisplay((position << celReps) | (position >> celReps));
        delay(50);                    //Adds a small animation when
                                        the player gets a point
      }
    }
    uint16_t randomInt = rand() % 15; //Choosing new random bar
    objective = 2 << randomInt;       //Making the random number a power of 2
```

```
  }
  while (digitalRead(reset) != 1 && score == maxScore) {
      //If the score == maxScore and the reset button is not pressed,
    setDisplay(65535);                //this will loop
    delay(100);                       //65535 is the decimal number that will
                                        turn on all LEDs when displayed
    clearDisplay();
    delay(100);
  }
  if (digitalRead(reset) == 1 && score == maxScore) {     //Resets score once reset
                                                            button is pressed
    score = 0;
  }
  setDisplay(objective | position);   //Displays the player and random bar at the
                                        same time
  timing++;                           //Increments the timer
  delay(duration);                    //Delay to increase duty cycle
}

void clearDisplay() {                 //Only ever need to clear the full screen so
                                        adding an integer
  for (uint8_t reps = 0; reps < 16; reps++)    //input is somewhat redundant
    write(0);                         //Writes 0 16 times and then latches
  setLatch();
}

void write(uint8_t bit) {             //Can write either 0 or 1 to the registers,
                                        will NOT latch
  if (bit == 0) {
    digitalWrite(clock, 1);           //All you need to do to write a 0 is toggle
                                        the clock pin
    digitalWrite(clock, 0);
  } else {
    digitalWrite(data, 1);            //Turns on the data pin, then the clock, and
                                        then turns them
    digitalWrite(clock, 1);           //both off
    digitalWrite(data, 0);
    digitalWrite(clock, 0);
  }
}

void setLatch() {                     //Turns the latch pin on and then off
  digitalWrite(latch, 1);
  digitalWrite(latch, 0);
}

void setDisplay(uint16_t displayNum) {  //This function works by accepting a 16 bit
  clearDisplay();                       number and mapping each bit to an LED.
  uint16_t mask = 0b1;                  // It is LSBF (least significant bit first).
  for (uint8_t i = 0; i < 16; i++) {
    (mask & displayNum) >= 1 ? write(1) : write(0);  //Ternary statement to turn on
    mask = mask << 1;                                 corresponding LEDs
  }
  setLatch();                                       //Latches at the end
  delay(duration);                                  //Adds delay to stop flickering
}
```

## Media

Project Video: https://youtu.be/dgIS07pN0mU

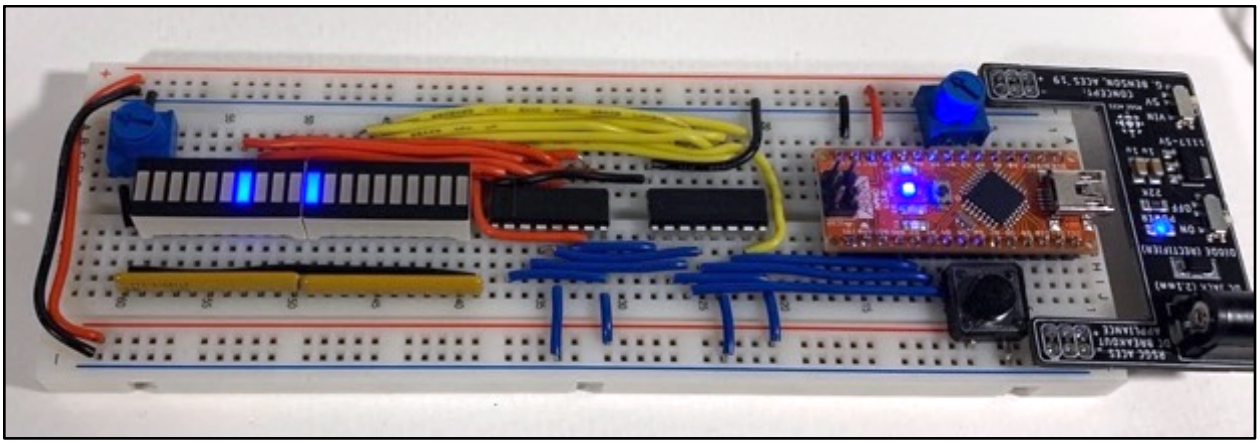
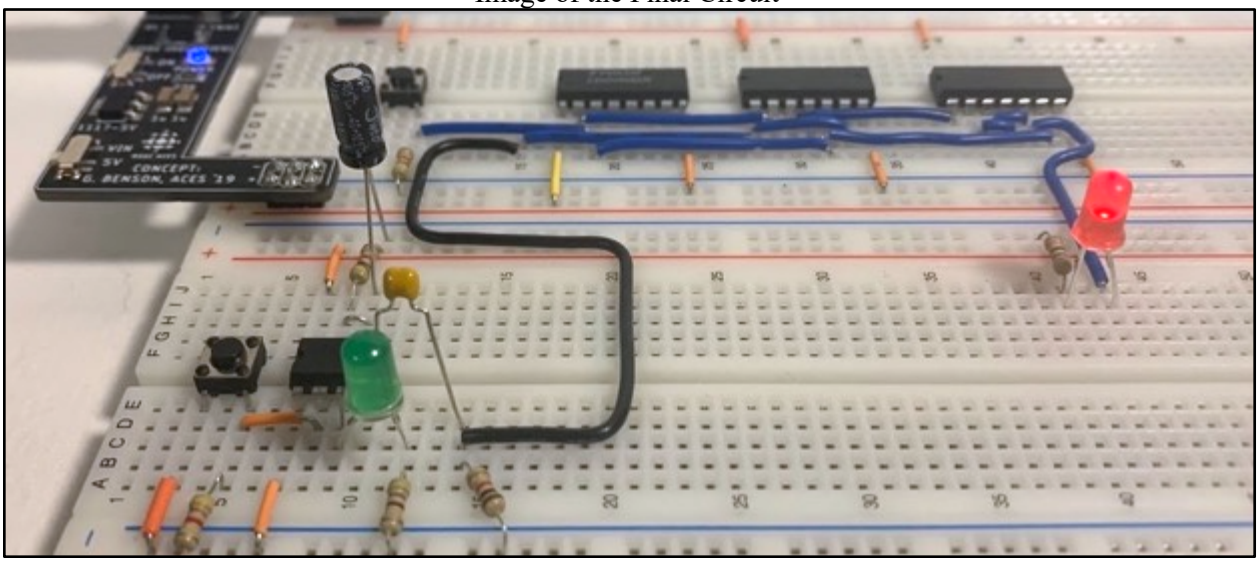


Image of the Final Circuit



Full Sized Image of D Flip-Flop

## Reflection

I personally found this DER really interesting. The integration of software into our previously hardware-only circuits offered much more room for creativity than normal and the project instructions recognized that by granting a great deal of freedom. However, this project also got me a bit worried. When picking my ISP I chose to base it around an 8x8 LED matrix, which I was planning to run through two daisy-chained shift registers. We have now covered shift registers in class and I know that our next project is using shift registers to drive LED matrices. I am worried that my ISP will end up being quite similar to the DER coming up in a month or so. Apart from those worries, I feel the shift register project went well. The NCC software practice definitely helped. I used many of the concepts that were new to me for my game. After completing this project, I took a look back at the code for a voltmeter that I made the weekend we got our Arduinos and was surprised by the dramatic increase in code quality between the two. Still, I of course have room to improve and would like to focus on using fewer lines to do the same thing in my future programs.

# Challenge 1.0 Sensor Monitoring and Display

## Purpose

The purpose of challenge 1.0 is to apply the skills learnt in class over the past few weeks, namely mapping and constraints. The end application is to display temperature readings from the TMP36 through the Morland Bargraph.

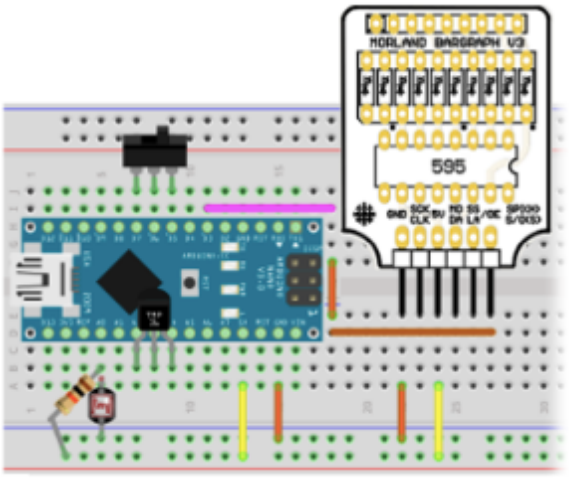
## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Challenge1/Challenge1.docx
Constrain Function: https://www.arduino.cc/reference/en/language/functions/math/constrain/
Map Function: https://www.arduino.cc/reference/en/language/functions/math/map/

## Procedure

### Hardware

The components of challenge 1.0 can be broken down into two main sections: the hardware and the software. Beginning with the hardware, the circuit is quite simple. Two voltage dividers are used as inputs into the Arduino Nano microcontroller. One divider is controlled by a TMP36 temperature sensor (in this case a 103 thermistor and a 10k Ω fixed resistor due to a parts mix-up) and the other is controlled by a light-dependent resistor (LDR) and another 10k Ω fixed resistor. These two inputs go into the Arduino and the temperature is then displayed on a bar graph, as seen to the right. The brightness of the bar graph can be controlled by exposing the LDR to either more or less light. A slide switch controls the mode of the display (between dot and bar).




As seen above, the output of the Arduino goes into a PCB. This is the Morland Bargraph V3 (MBV3). The MBV3 is a soldered version of one-half of the previous project, 2.2. Essentially, the MBV3 is a breadboard appliance that makes wiring and testing shift register projects easier, as well as minimizing pin usage. Due to the traffic light being omitted from this year's DER, this is the first encounter with an appliance. An appliance is not a technical term but refers to any soldered device meant to be directly inserted into an Uno or breadboard. Since the Arduino Uno has no shared "rail" with 5V, GND, and IO pins, the MBV3 is limited to breadboard usage due to its current draw being too great for IO pins.

| **Parts Table** | |
|---|---|
| Description | Quantity |
| Arduino (ABRA) Nano | 1 |
| Slide Switch | 1 |
| 104 Thermistor | 1 |
| Light Dependant Resistor | 1 |
| 10 kΩ Fixed Resistor | 2 |
| Morland Bargraph V3 | 1 |
| Assorted Jumper Wires | 8 |

## Software

The real meat of this challenge is in the software. It is essentially a culmination of most skills learned in class until this point. Something helpful when tackling software problems is to write an algorithm, a series of steps of what has to happen for the desired outcome. To the right is one for this challenge. Clearly, the algorithm does not use actual coding language, but having it at least appear to be code can be helpful in planning. As seen in the algorithm, the general flow of what needs to happen is as follows: First, read the temperature data from the TMP36. Then convert that data to the temperature in Celsius before limiting the possible values to usable numbers (between zero and seven corresponding to the eight output pins of a shift register). Next, depending on the mode of the slide switch, either set the display to the bar value or the dot value. Finally, turn the display on for the HIGH time of the duty cycle and off for the LOW time, all controlled by the LDR.

```
//Algorithm
loop() {
  read(tempData);
  convert(tempData to tempCelcius);

  limit(tempCelcius to between(0,7));

  if (switchMode == dot) {
    display(dot);
  } else if (switchMode == bar) {
    display(bar);
  }
  read(LDRData);
  convert(LDRData to dutyCycle);

  setDisplay(on);
  delay(timeOn);
  setDisplay(off);
  delay(timeOff);
}
```

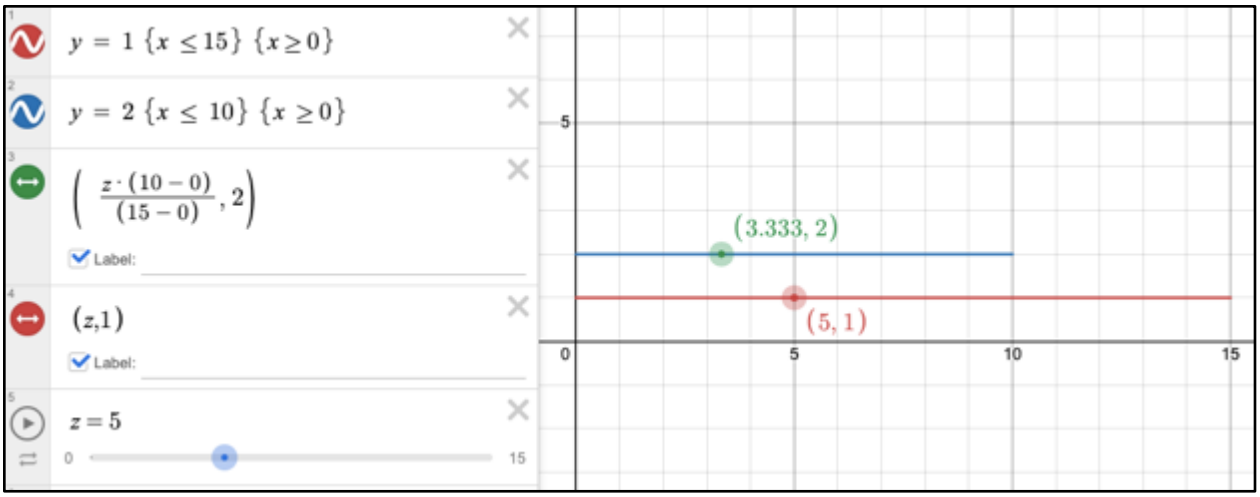
## Constraints & Mapping

Some of the wording in the previous section was ambiguous, such as limit and convert. These two words in particular have built-in functions in the Arduino IDE. Limit and convert are called constrain and map respectively within the IDE. The `constrain(x, a, b)` function takes input x, and if it is greater than a and less than b, returns x. If x is less than a, x is set to a. If x is more than b, x is set to b. Essentially, `constrain(x, a, b)` takes x and makes sure it is included in the domain of [a, b]. Interestingly, x cannot be a negative value in some situations. This is most likely due to `constrain()` using unsigned integers in the function's source code.

Next, the map function is a bit more complicated. The purpose of mapping is to take a number from one range and change it to the corresponding number in another range. The `map(x, a, b, c, d)` has five inputs. X is the number within the first range, a & b and d & c are the respective minimum and maximum values of the first and second ranges. To solve for the corresponding x in the second range (x'), comparison is used. The ratio of x to the difference of the first range will be equal to the ratio of x' to the difference of the second range. From there, if both sides of the equation are multiplied by the difference of the second range, the result it $x' = \frac{x \times (d - c)}{(b - a)}$. Strangely, the Arduino IDE uses the formula $x' = (x - a) \times (d - c) \div (b - a) + c$ which does the same thing as the first example but has added steps. Subtracting a from x results in a difference of c when (x – a) is passed through the rest of the formula, so it is technically correct. However, (x – 5a) at the beginning and + 5c at the end would also be correct. The first formula is the simplest form.

$$\frac{x}{(b - a)} = \frac{x'}{(d - c)}$$

$$\frac{x * (d - c)}{(b - a)} = x'$$




Above is a screenshot of Desmos. The red and blue lines represent two ranges. The red is the original range, and the point 5 is being mapped to the blue range. The formula successfully outputs 3.33, which is one-third of 10 (the blue range). The question now is why? Where is the need for constrain and map?

The TMP36 temperature sensor has an operating range from -40 to 125 degrees Celsius. However, the reading on the analog pin of the Arduino is not in the range of -40 to 125. The TMP36 increases voltage by 10 mV per degree Celsius. That means the range of the TMP36 is $10\ mV \times \big(125 - (-40)\big) = 1650\ mV$, or 1.65 V. However, the Arduino also does not read in voltage. The final range of the TMP36 is 337.59 ($1023 \div \frac{1.65\ V}{5\ V} = 337.59$). According to some sources, there is a y-intercept of 38 mV making the range a bit higher. Either way, the desired number for the temperature reading is close to `map(x, 0, 337.59, -40, 125)`. However, since the display should only show the temperature as eight possible values ([22, 29]), that new value then has to be constrained. After those steps, the reading will be converted to a value between 22 and 29.

Admittedly the presented formula for the map function is a bit convoluted. In reality, it is only a linear relation. X' will always equal mx, where m is the slope $\frac{range_2}{range_1}$. However, a two-dimensional linear relation is a somewhat confusing way to represent the relation of two one-dimensional number lines.

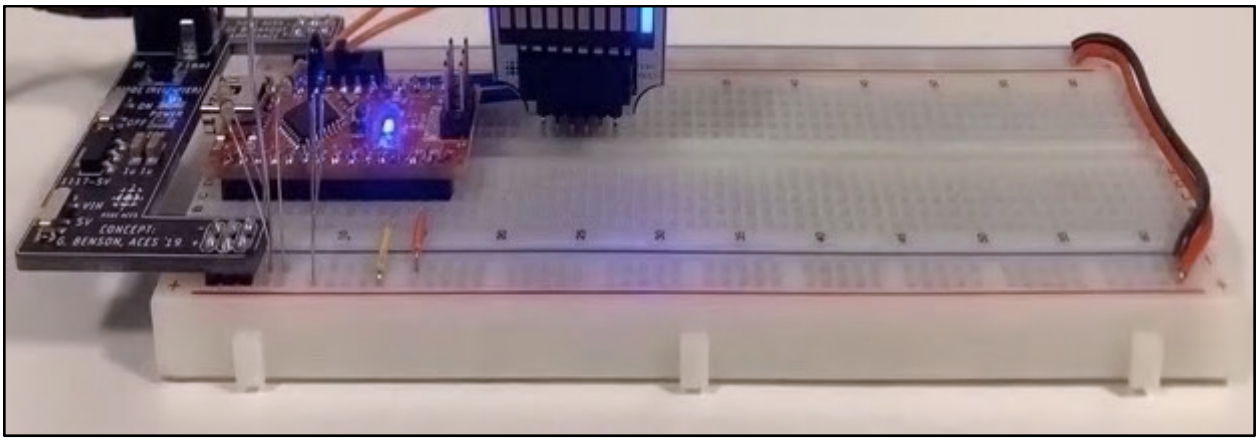
## Code

```
// Project   : Sensor Monitoring and Display
// Purpose   : To display temperature readings on the MBv3 in either bar or dot mode
// Course    : ICS3U
// Author    : Liam McCartney
// Date      : 2022 11 19
// MCU       : 328p
// Status    : Working
// Reference : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Challenge1/Challenge1.docx

#define clock 10            //Defining shift register pins
#define data 9
#define latch 3
#define outputEnable 2
#define duration 1          //Not useful to change, but it is possible

uint16_t tempRead;
uint16_t tempC;             //Declaring variables
uint16_t dutyCycle = 0;
```

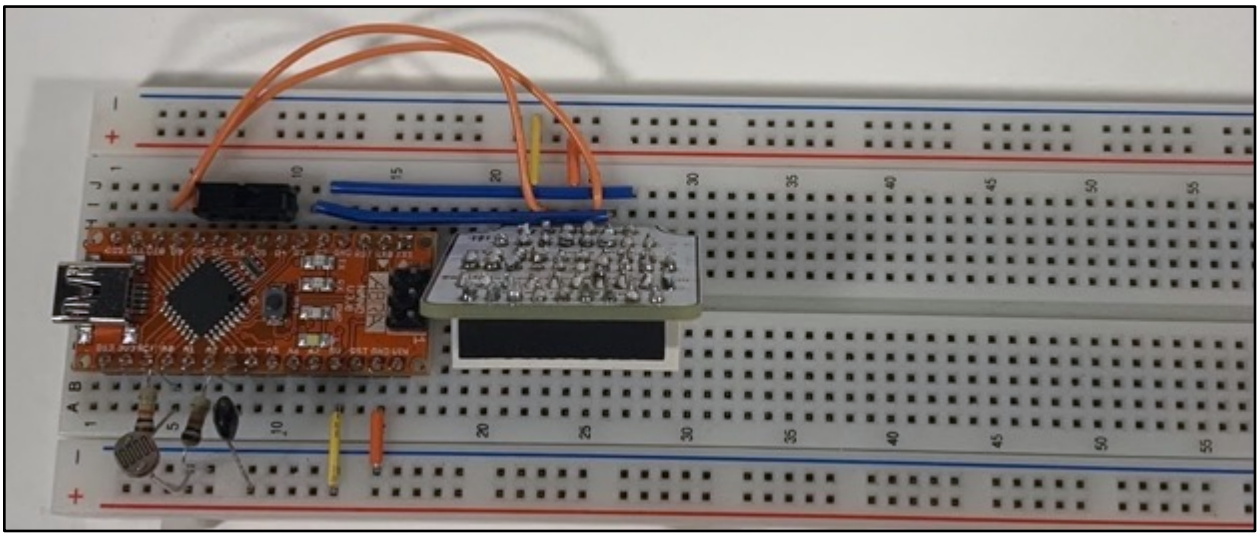
```
void setup() {
  Serial.begin(9600);
  while (!Serial)
    ;
  Serial.println("Hello World");

  pinMode(data, OUTPUT);
  pinMode(latch, OUTPUT);
  pinMode(clock, OUTPUT);
  pinMode(outputEnable, OUTPUT);
  digitalWrite(outputEnable, 0);
  clearDisplay(1);
  setLatch();                          //Setting up the shift register

  pinMode(7, OUTPUT);                  //Slide switch mode control pins
  pinMode(5, OUTPUT);

  digitalWrite(7, 1);                  //Setting one switch pin high and
  digitalWrite(5, 0);                  //The other low
}

void loop() {
  tempC = map(analogRead(A3), 0, 1023, -50, 100);          //Mapping from input range
                                                             to operating range
  uint16_t constrainTemp = constrain(tempC, 22, 29);       //Constrain to between the
                                                             max and min temp
  constrainTemp = tempC - 22;                              //Converting to a useable
                                                             value

  digitalRead(6) == 1 ? shiftOut(data, clock, MSBFIRST, (1 << constrainTemp)) :
shiftOut(data, clock, MSBFIRST, (1 << (constrainTemp++)) - 1);
  setLatch(); //The above line reads the mode switch and either shifts out the bar
                or dot value

  digitalWrite(outputEnable, 1);                           //Turns off display
  delay(duration * (1023 / (1023 - dutyCycle)));           //Waits for LOW portion of
                                                             duty cycle
  digitalWrite(outputEnable, 0);                           //Turns on display
  dutyCycle = analogRead(A0);
  delay(duration * (1023 / dutyCycle));                    //Waits for HIGH portion
  }                                                          of duty cycle

void clearDisplay(uint8_t regNum) {
  for (uint8_t reps = 0; reps < regNum; reps++)
    shiftOut(data, clock, MSBFIRST, 0b00000000);           //Shifts out 0 for reps
                                                             number of registers
}

void setLatch() {                                 //Turns the latch pin on and then off
  digitalWrite(latch, 1);
  digitalWrite(latch, 0);
}
```

## Media

Challenge Video: https://youtu.be/PmGsigoBljM



Front View



Top View

## Reflection

I think that the idea of spending more time on something that we had such a short time to complete in class is an interesting task. It was frustrating to debug the problems that I had in class within a few minutes of looking at the code without the time pressure, but rewarding nonetheless. As for the report, I tried something different with the mini code block for my algorithm. It is something that I sometimes actually do when planning out a project, but at the same time it is not real code and I do not know if it is better than another option, maybe such as the block graphics I have used in the past. With this being an optional assignment, I took the opportunity to experiment a bit. Besides that, I actually got very invested in the map function. I enjoyed my little trip to Desmos to find a better visual representation of the math. I also found it extremely odd that the Arduino IDE seems to add extra unnecessary steps to the math in their map function. In the end, I am pretty happy with this report. It ended up being more theory-based due to the focus on coding – something that is not tangible – but I did not mind that too much. However, I am looking forward to moving back to hardware with the LED matrix next week.

# Project 2.10c Short ISP: The Matrix Graphing Calculator

## Theory

This Independent Study Project (ISP), the Matrix Graphing Calculator (MGC), takes a linear relation between two potentiometer readings and converts it into a suitable input for a LED matrix. Using persistence of vision and multiplexing, the matrix displays a rasterized approximation of the inputted relation. As well as graphing the relation, the slope is also calculated with a denominator of one and displayed for the viewer on two seven-segment displays. In addition, the MGC explores new ideas introduced with ISPs, such as casing and permanence.

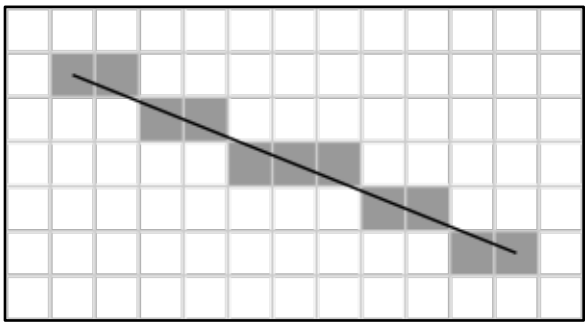
## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/ISPs.html#logs
Bresenham's Line Algorithm: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
Rasters & Vectors: https://www.acromedia.com/article/graphic-files-explained-vector-vs-raster
Persistence of Vision: https://en.wikipedia.org/wiki/Persistence_of_vision
Desmos: desmos.com
Manim: https://github.com/3b1b/manim

## Procedure

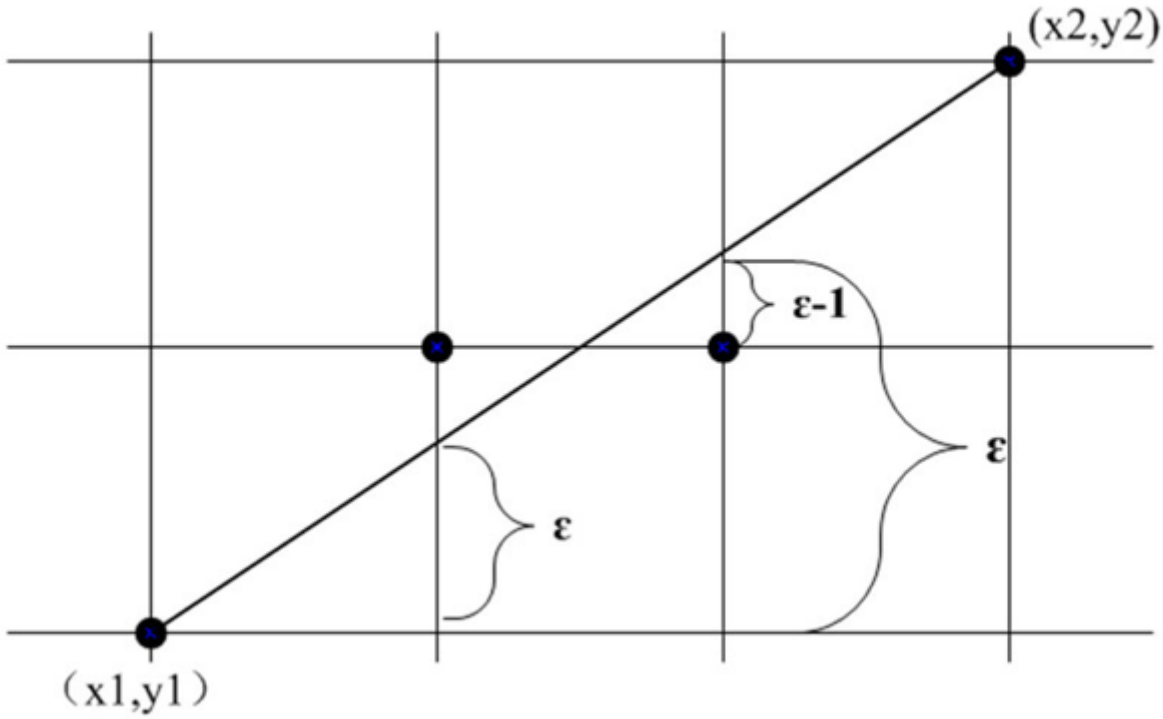
### Vectors, Rasters, & Line Algorithms


The main complexity of this project, as mentioned in the theory section, is the conversion of slope – a vector – into a pixelated image – a raster. Vectors and rasters are somewhat ambiguous terms with multiple meanings in different subjects, but in this case, the relevant meanings of vector and raster are concerning graphics. A vector is a type of graphic that can be scaled, transformed, rotated, or manipulated in any way while not losing any definition or quality. This is because vector graphics are mathematically defined in geometric shapes and lines. Even these geometric shapes are defined using relations, so in reality, vector images are a collection of different functions and relations. This differs from rasters, which are a collection of instructions for specific pixels on a display. Rasters are not defined by any math and therefore have a set definition of height and width. At the most basic level, rasters are stored serially in one-dimensional arrays. Storing an image like this has a file name called bitmap file or .BMP. However, most commonly rasters are stored in compressed forms with a cipher to decode them. Compression will not be relevant to this project, only bitmaps.



These two image formats are important because a slope is a linear relation, meaning it is a vector, while eight by eight LED matrices use pixels to display images, meaning that the slope will have to somehow be converted from a vector to a raster bitmap. An example of one such conversion is shown to the right. Luckily, the process of plotting a line onto a matrix is not untrodden territory, but nonetheless it will be the most intensive part of the MGC.

By far the most famous technique for rasterizing a line is Bresenham's line algorithm. Admittedly, it is overly complicated and somewhat redundant for this purpose. To begin with, Bresenham's algorithm draws the approximation of a line linking two points on a grid (matrix), which in this scenario is unnecessary since the MGC will only graph from (0, 0), or the bottom left corner of its display. However, it is still a great starting point to begin thinking about how to create a line plotter from scratch. Shown below is a visual representation of Bresenham's algorithm for slopes less than one. A key thing to understand when dealing with Bresenham's algorithm is that for any pixel, there are only two options. You can only draw the next pixel at the current y-value, or one higher than the current level (assuming the slope is less than one). You already know that the x-value at the next pixel will be the current value plus one, so really, all that the algorithm has to determine is if the next pixel should be one higher or on the same level as the current pixel.




To do this, Bresenham employed what he called the error between the raster approximation and the vector. He figured out that the error for any pixel is equal to the previous error plus the slope of the line (the error starts at 0 for the first pixel, which is placed on $(x_1, y_1)$). This error is shown with the lowercase epsilon (ε) on the graph. Bresenham discovered that if ε > 0.5, the next pixel will be one pixel higher than the current pixel. If ε < 0.5, the next pixel will be at the same height as the current pixel. If the next pixel is moved up, the error is changed to ε - 1 to account for the vertical shift.

As mentioned, this version of Bresenham's algorithm only works if the slope (m) is less than one, and the same will be true for the algorithm of the MGC. This is because the concept of the next pixel only being one higher or the same height will only be true for the octant where 0 < m < 1. Take m = 2 for example. The next pixel will never have the same height as the current pixel. However, the x of the next pixel will only ever equal the current x or current x + one. This can be expanded to when m > 1, the next x-value equals the current x-value if the x-error is less than 0.5, but will equal current x + 1 if the error is greater than 0.5.

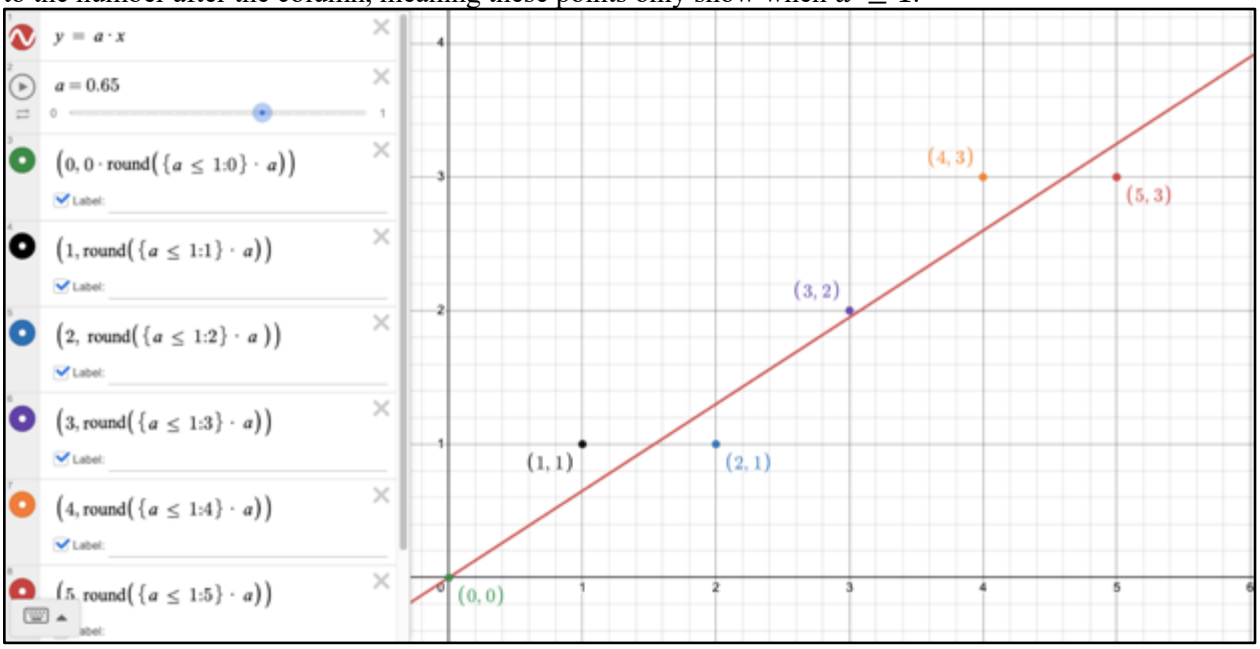
```
slope = ((y2 - y1) / (x2 - x1));
error = 0;
drawPixel(x1, y1);

for (uint16_t x = x1; x < x2; x++) {
  error = error + slope;
  if (error > 0.5) {
    y = y + 1;
    error = error - 1;
  }
  drawPixel(x + 1, y);
}
```

To the right is Bresenham's line algorithm written out in pseudo-code. This version only works when m > 1, but that is unimportant at the moment. With Bresenham's algorithm explored, an original algorithm can now be constructed. Some parts will be essential to keep, such as what to do if the slope is greater than or less than one, but some other parts can be changed. Firstly, as already mentioned, starting and ending points will be unnecessary. In fact, since the MGC graphs a line based on a slope, it will not even know the endpoint of the line. Also, the error part of the algorithm is important, but the same outcome can be achieved by manipulating variable types to do rounding for us. The final and biggest change is that this new algorithm will be solving for either x or y, not determining whether or not to increment the previous value.

```
float slope = rise / run;

if(slope < 1) {
  for (numColumns) {
    int y = columnNum * slope + 0.5;
    drawPixel(columnNum, y);
  } else {
    for (numRows) {
      int x = rowNum * 1/slope + 0.5;
      drawPixel(x, rowNum);
    }
  }
}
```

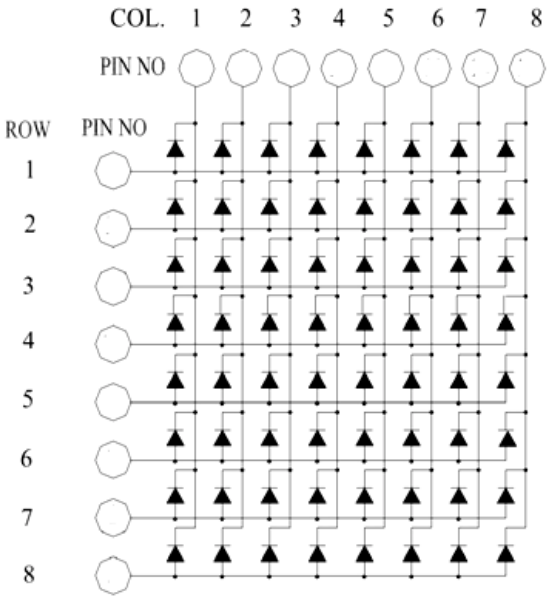
The new algorithm is located on the right. It better suits the needs of the MGC than Bresenham's algorithm. The input is a slope, not two points, but the output is still a set of coordinates. Instead of using a running error, this algorithm takes advantage of the necessary use of float variables (slopes are by nature not integers, the math would be incredibly complicated to avoid floats). When setting an integer equal to a calculation involving a decimal in c++, the integer is simply set to the value to the left of the decimal place without any rounding. However, if 0.5 is added to the float, any value of the right that would have rounded to one is increased to some value greater than one, spilling over to the ones digit and effectively rounding the float to an integer.

Below is the new algorithm approximating a slope of 0.65. A quick explanation of Desmos formatting may be necessary. Take $\{a \leq 1: 1\}$. This means that if $a \leq 1$ then the expression in braces is equivalent to the number after the column, meaning these points only show when $a \leq 1$.



## LED Matrices

After deciding on how to find the approximation of a line in the previous section, it is time to move on to the next idea: persistence of vision. However, to understand persistence of vision and why it is needed, it is first necessary to understand LED matrices. To the right is a picture of a single eight by eight LED matrix. Every single white circle on the surface is an LED, totalling 64 LEDs on the whole matrix. As mentioned, the slope will be graphed onto a matrix such as this one. To be able to write code for a matrix that will display a graph, it is important to know how LED matrices work under the hood because it is both incredibly helpful and also presents a problem.

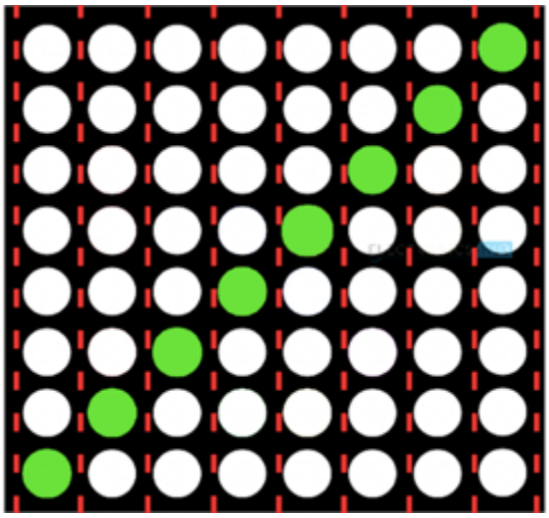



To the right is a schematic of a matrix. It is not actually too complicated; it is essentially just 64 LEDs connected by eight common anodes and eight common cathodes. This cuts down the necessary IO pins on the Arduino down to 16 from the 64 needed for 64 lonesome LEDs (in reality, shift registers will cut down the necessary pin number to only three for all 64 LEDs). To turn on any one pixel, the corresponding row must have a voltage input and the corresponding column must have ground present. All other rows should be low and all other columns should be high to assure that the only light LED is the one desired by the user. This will present voltage at the common cathode of the desired LED's row and ground at its base, creating a potential difference for current to flow. However, since all other rows will be low and columns will be high, no other LEDs will have a potential difference. This system essentially assigns a coordinate point to every LED on the matrix, which is perfect for the needs of the MGC because the previously explored algorithm solves for coordinate points.

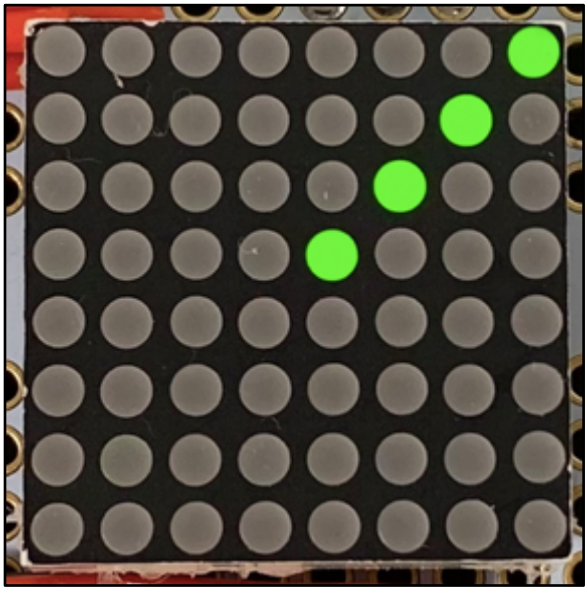
However, there is one problem. The MGC will have to turn on multiple LEDs. Say, for example, the current slope is one. That would mean that each light pixel would be diagonal to both its neighbours. To graph this slope with the current method, all rows would be turned high and all columns would be turned low because there would be one pixel on each row and each column. However, the effect of this would be to turn on all pixels with a high row and low column. In this case, graphing a slope of one would result in every single LED in the matrix turning on. Clearly, this is not the desired outcome. The solution here is persistence of vision.

## Persistence of Vision

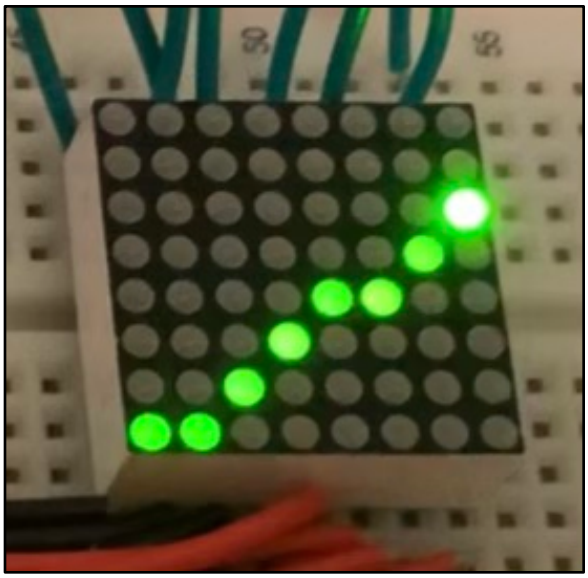
Persistence of vision is technically a term for an optical illusion where we perceive past visual stimulants as being present for a short period after they are removed from our vision. An extreme example of this is looking at the sun; you can visualize it in your mind after closing your eyes and looking away for a few minutes. The real name for the technique that is employed to drive LED matrices is multiplexing, and it makes use of the phenomenon of persistence of vision. Multiplexing consists of a "scanning" action of either rows or columns of the matrix and turning on the corresponding rows for the current scanned column and vice versa when scanning rows.



To the right is a diagram of the slope of one. For every column, there is only one pixel that should be on (this is the same for every row, but for any other slope one of the two would not be true; hence the need for $m > 1$ and $m < 1$ conditions in the algorithm). To display just the desired pixels by multiplexing, only one column is turned on at a time. Beginning with column 1, we turn on the necessary row (row 1) and delay for a short time. After the delay, column 1 is turned off and column two is turned on, where the correct rows are also turned on (row 2). There is again a short delay before column two is turned off before the process continues with column 3, then 4, etc. If the delays are short enough, the human eye will perceive all pixels as being on all the time and the matrix will appear to display a slope of 1 at all times.



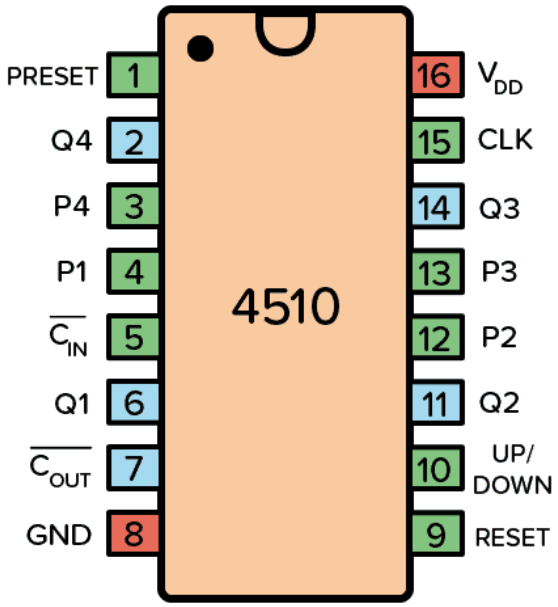
Using a newer smartphone and lots of light, it is possible to capture a picture where the full display is not all on at once. This is due to the extremely fast shutter speed of modern smartphones, but even so, there are multiple columns illuminated at once. This photo was taken at night with low natural light. It is possible that the same photo during the day would have more background light and therefore a faster shutter speed, resulting in fewer illuminated pixels. This makes it quite difficult to capture pictures of devices using persistence of vision and is why lights often appear to flicker in videos. Since the human eye sees at only around 30-60 "frames per second" (this is a misuse of the term frames per second, but no better explanation exists), the delay can actually be quite long relative to the world of electronics. In fact, in practice it is almost impossible to notice persistence of vision unless the delay is above 10ms for each column (assuming that there are eight columns and the device is stable). Interestingly, delays even below 5ms are noticeable if the device is moved around relative to the viewer.



Finally, for persistence of vision, the image on the right demonstrates why somewhat counterintuitively, a longer delay is better than the shortest possible delay. The code running the matrix in this image is optimized for the shortest delay possible to minimize flickering, but the obvious issue is that in this image the top right pixel is much brighter than the other pixels. This is vastly different from the other image where all pixels are virtually identical, and it is due to three lines of code in the loop section in addition to the function that displays the graph. These lines do not delay; their execution time is simply so long in relation to the short delay between columns that the top right pixel is left on for a proportionately long time compared to the other pixels. This results in an off-putting pixel that is too bright. To solve this, more delay is added to the time between columns while multiplexing, making the ratio of the execution time of the three lines before displaying to the delay between columns much more negligible.

## Displaying Slope

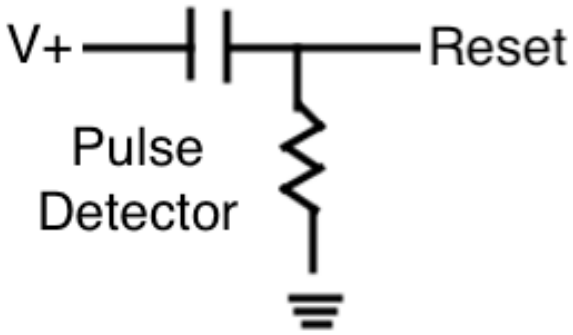
As described in the theory section, the MGC also displays the slope of the line on two seven-segment displays with a denominator of one. To do this, two 4510 and 4511s are employed to drive the displays. It would of course be possible to daisy-chain the shift registers used to drive the matrix, but shift registers have already been explored. Instead, the 4510 and 4511 method uses only one pin to drive a single seven-segment display. In a circuit with no shift registers this is more pin-efficient, but more importantly, it presents a challenge and an opportunity to learn. As both these chips were used previously in the Counting Circuit, there will only be a quick recap of their function. The 4510 IC is a BCD counter with a range of [0, 9]. It advances its count by one on every rising edge that the clock pin experiences, and resets its count on the rising edge of the reset pin. The 4511 takes the output of the 4510 and decodes it into an input for seven-segment displays.




Driving a seven-segment display with this setup is fairly straightforward. To set the display to the desired number, pulse the clock pin the number of times necessary to change the count to the desired number. Determining the necessary number of pulses is done in software. The 4511 will decode the BCD number of the 4510 and display it to the seven-segment display, so the only connection to the Arduino is one wire from the clock to an IO pin.

There is, however, one problem. Through testing, the 4510 appears to save its count when its power source is removed (interestingly, this is not mentioned on its datasheet). This presents an issue: if the circuit is turned off with the on/off switch, or the battery/USB port is unplugged, the circuit will lose power, causing the Arduino to forget the saved state of each display while the 4510s will remember their counts. This means that when the circuit is turned back on, the displays will be out of sync with the Arduino, causing the displays to be inaccurate.

To solve this there are two approaches. You could either have the Arduino keep its stored value for the displays, or change the displays to a predictable number. The first option would be done in software using EEPROM, which is not inherently a bad thing, but the second option can be done using only hardware and is therefore the winner. The strategy employed is to reset the count of both 4510s every time the circuit turns back on using a pulse detector. This circuit "detects" when the power comes back to the circuit and will turn the reset pins of both 4510s high when power is restored. The Arduino can then be sure that when it begins the program, the counts will both be at zero. This circuit is virtually the same as the rising edge detector in the earlier shift registers report (Project 2.2), an explanation of the circuit's function is located in that project.




## Software

After understanding the task at hand and how all the pieces work, it is time to write an algorithm. However, there is one final thing to do before that. The line algorithm from earlier works well, but it does not produce outputs suitable for the matrix to display. The coordinate points have to be translated into a byte representing the columns and another byte representing the rows. Luckily, this is easy enough. The possible coordinate values range from 0 – 7. To convert this to a byte, all that has to happen is 1 has to be bit-shifted y times to find the row byte. For the columns byte, we need to do the same thing (1 << x), but then either NOT the output of XOR the output with 255. Both have the same effect of changing the output from one 1 and seven 0s to seven 1s and one 0, which is necessary because the significant bit in this case should be a 0, not 1.

To the right is the final algorithm for the whole MGC. The general flow is as follows: 1. Calculate the slope of the line. 2. If statement to use the right process for different slopes. 3. Solve for the coordinates of the pixel. 4. Use the above-mentioned strategy to turn the coordinates into a column byte and a row byte. 5. Display the pixel on the screen (by shifting out the bytes), and wait for a short time. 6. Determine the number of times to pulse the clock pins on the two 4510 chips. 7. Pulse the clock pins the determined number of times. 8. Repeat.

The seven-segment displays must be altered during the coordinate-solving phase to keep the ratio of time on for average LED to time on for the first LED equal (uniform brightness).

```
float slope = rise / run;

if (slope < 1) {
  repeat(numColumns) {
    int y = columnNum * slope + 0.5;
    determine(xByte, yByte);
    display(xByte, yByte);

    find(clockPulseNum);
    pulseClock(clockPulseNum);
  }
  else {
    repeat(numRows) {
      int x = rowNum * 1/slope + 0.5;
      determine(xByte, yByte);
      display(xByte, yByte);

      find(clockPulseNum);
      pulseClock(clockPulseNum);
    }
  }
}
```

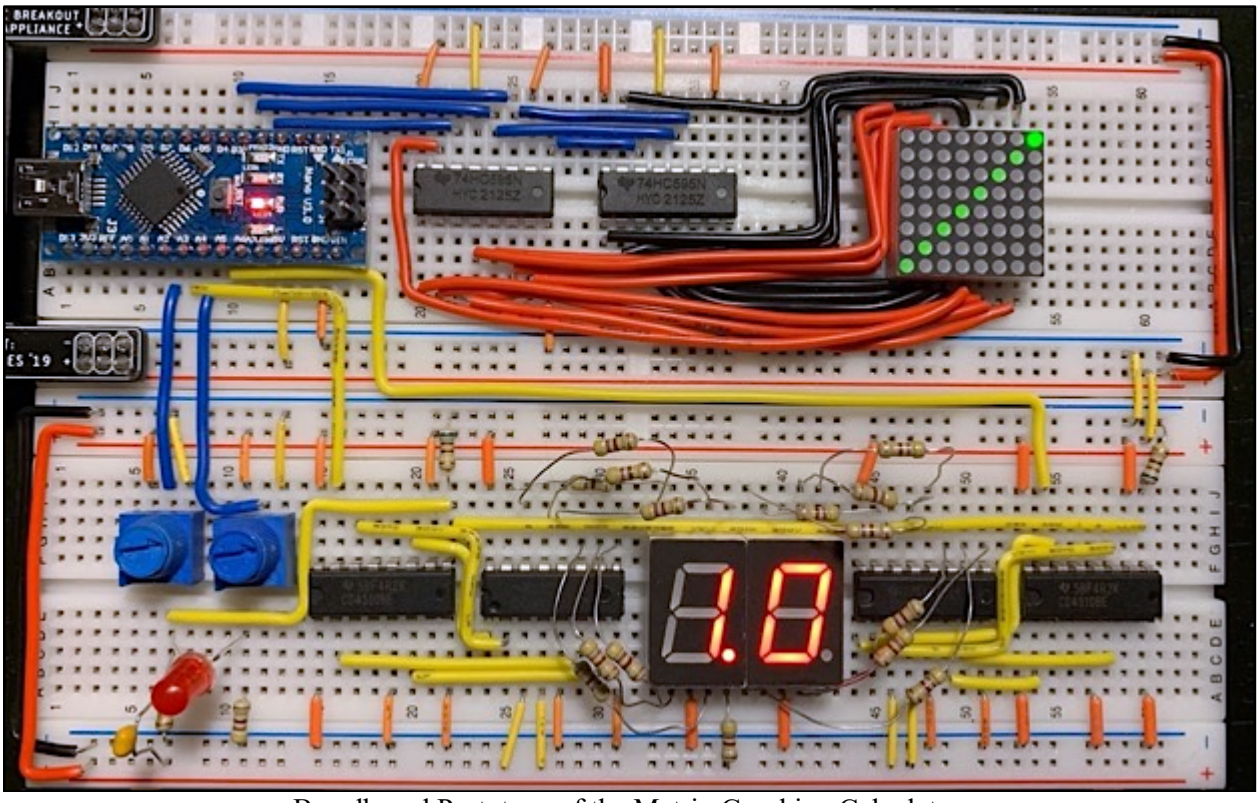
## Hardware

### Breadboard Circuit

The breadboard prototype takes up almost two full boards with the onboard Nano. It is missing certain features that the permanent circuit has, such as an on/off switch. Two daisy chained shift registers are located in the center of the first board which have not been explicitly mentioned yet. The column and row bytes that the code determines are shifted out onto these two shift registers. The red wires deliver current to the row pins, hence their colour. The black wires are the ground wires for the columns. Something quite frustrating is that this is the only way to wire the shift registers to the matrix due to its very strange and seemingly arbitrary pinout. This pinout causes wires to cross over each other and looks rather displeasing. Also, on the prototype, there is a red LED to show the reset of the two seven-segment displays. This was removed in the permanent version due to the pulse being so quick that it was incredibly hard to see.

**Parts Table**

| Description | Quantity |
|---|---|
| Arduino (ABRA) Nano | 1 |
| 74HC595 Shift Register | 2 |
| 8x8 LED Matrix | 1 |
| 10 kΩ Potentiometer | 2 |
| 470 Ω Fixed Resistor | 17 |
| 680 Ω Fixed Resistor | 3 |
| 0.1 µF Capacitor | 1 |
| 4510 BCD Counter IC | 2 |
| 4511 BCD Decoder IC | 2 |
| Seven-Segment Display (cc) | 2 |
| Jumper Wires | ** |



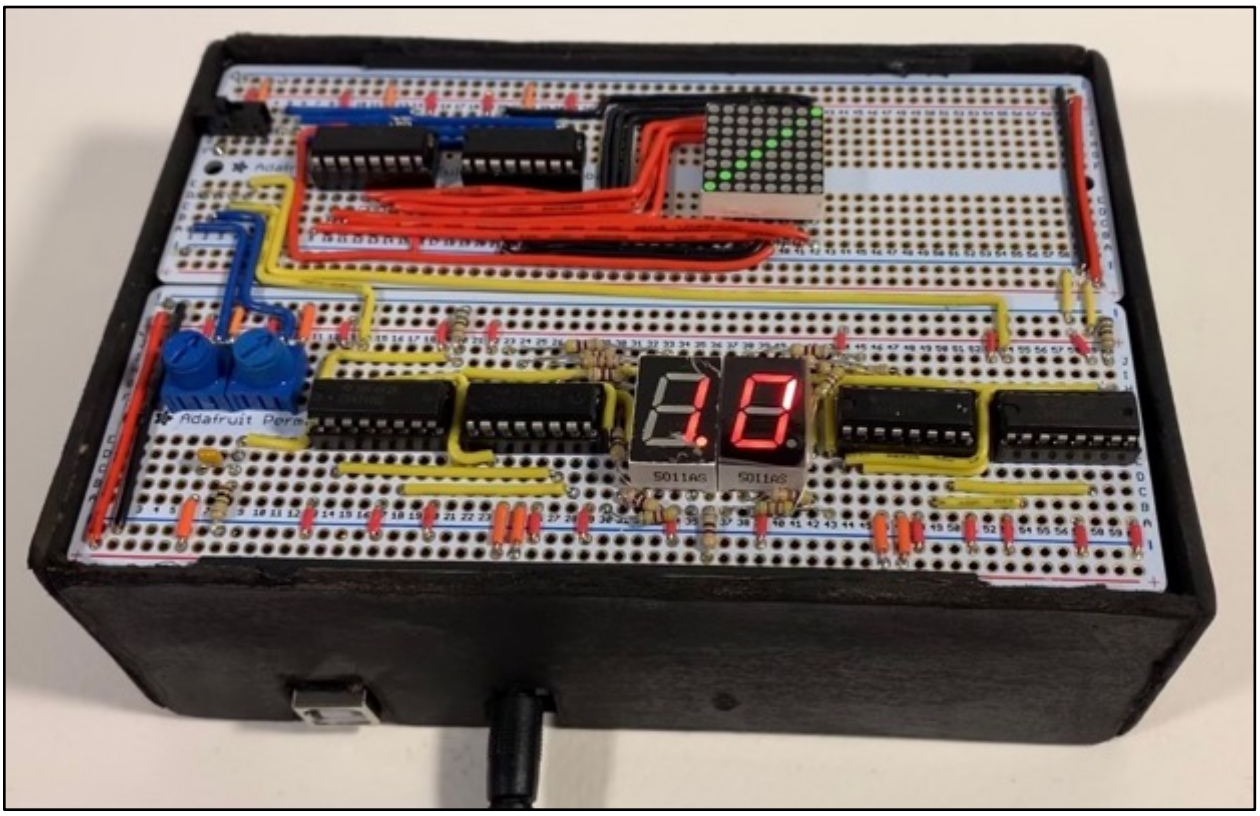
Breadboard Prototype of the Matrix Graphing Calculator

## Permanent Circuit

The permanent version of the MGC is quite similar to the breadboard prototype in terms of circuit architecture. Other than that, they are quite different. The permanent version is soldered onto Adafruit's Perma Proto boards. The case is made out of balsa wood and has then been spray-painted black. A wooden case was chosen over a 3D-printed one due to its reliability. Wood does not get jammed or run out of filament. The same cannot be said about 3D printers. Both the Uno and the Perma Proto boards are held in the case by a tight fit. They are easy to remove if you know how to do it, but will not fall out on their own. In fact, both the Uno and the Perma Proto boards have very little travel. The Uno is connected to the board with male headers and jumper wires. Male headers were soldered on from the top to act as "ports" for male-female jumper wires to attach to from the Uno below. Finally, power is provided through either the barrel jack as shown below or through the USB port, meaning the Uno can be programmed while in the case.

**Parts Table**

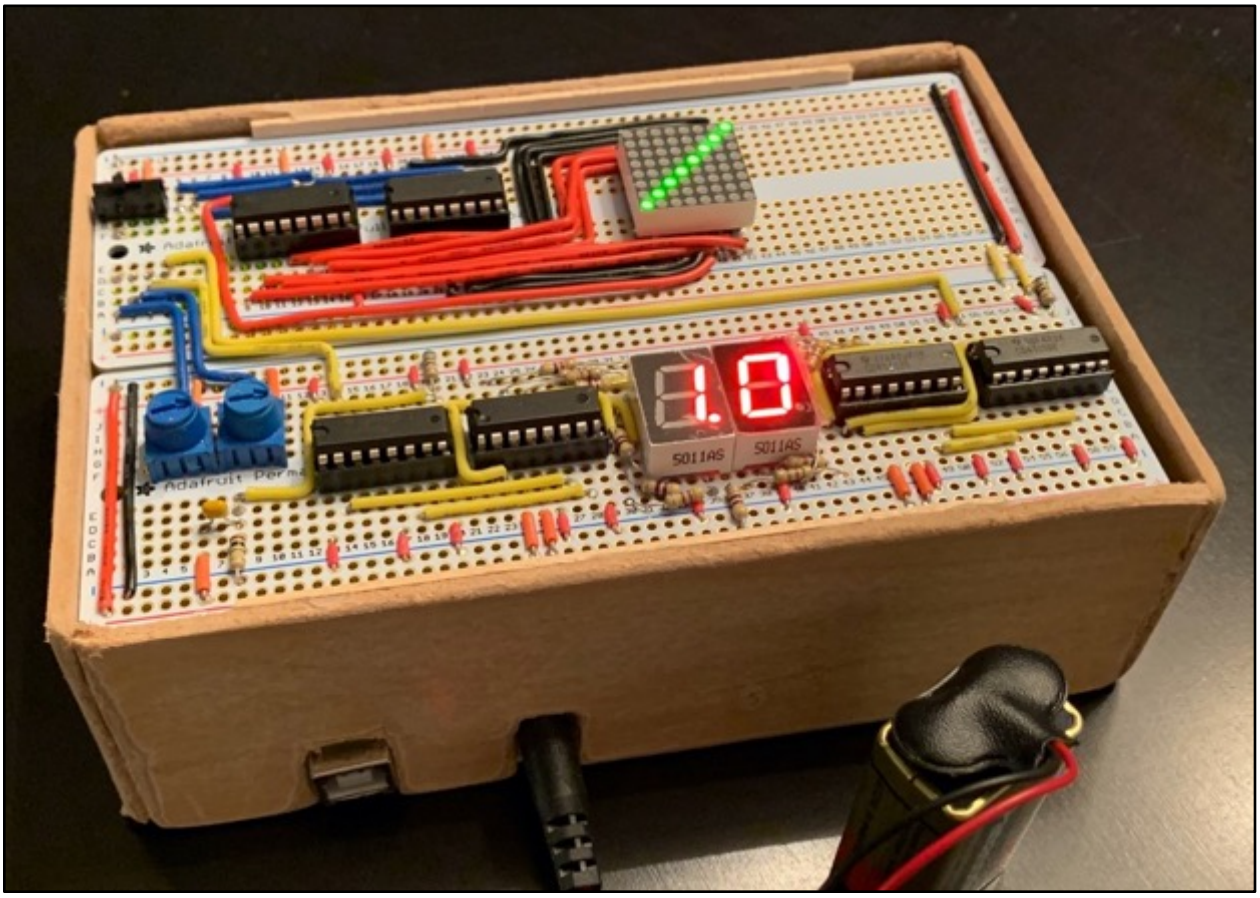
| Description | Quantity |
|---|---|
| Arduino (ABRA) UNO | 1 |
| 74HC595 Shift Register | 2 |
| 8x8 LED Matrix | 1 |
| 10 kΩ Potentiometer | 2 |
| 470 Ω Fixed Resistor | 16 |
| 680 Ω Fixed Resistor | 3 |
| 0.1 µF Capacitor | 1 |
| 4510 BCD Counter IC | 2 |
| 4511 BCD Decoder IC | 2 |
| Seven-Segment Display (cc) | 2 |
| Slide Switch | 1 |
| Jumper Wires | ** |



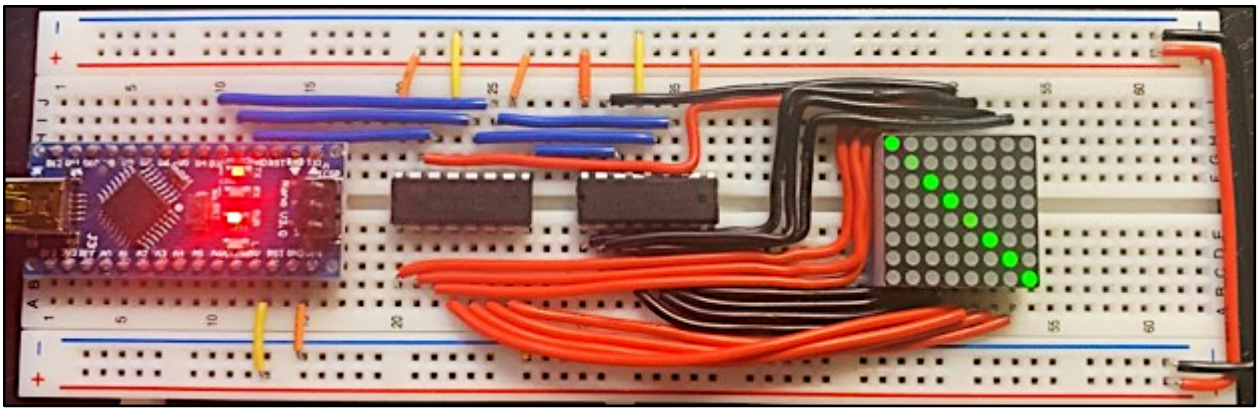

Permanent Matrix Graphing Calculator

## Media

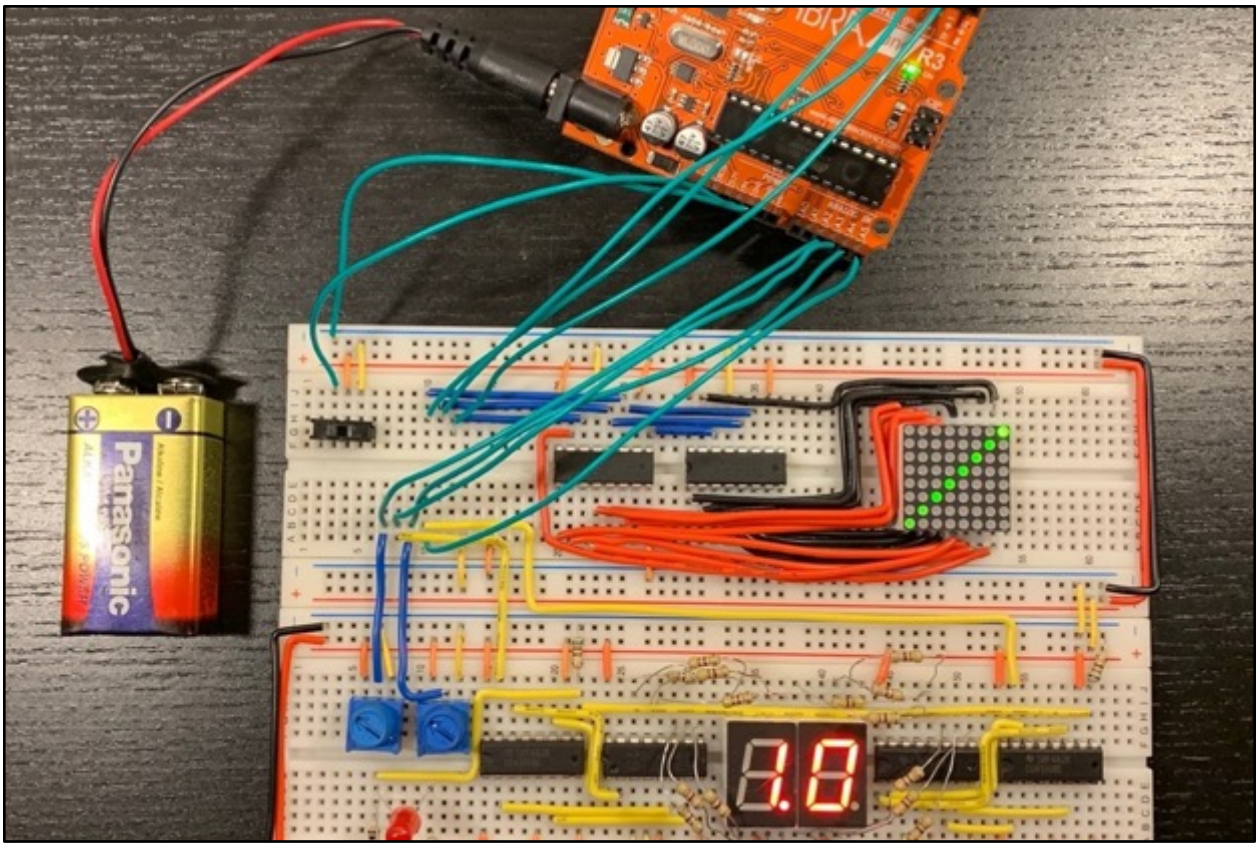
Project Video (2 minutes): https://youtu.be/795KpdAJDic
Full Length Project Video (4 minutes): https://www.youtube.com/watch?v=8zpBBVYscDs



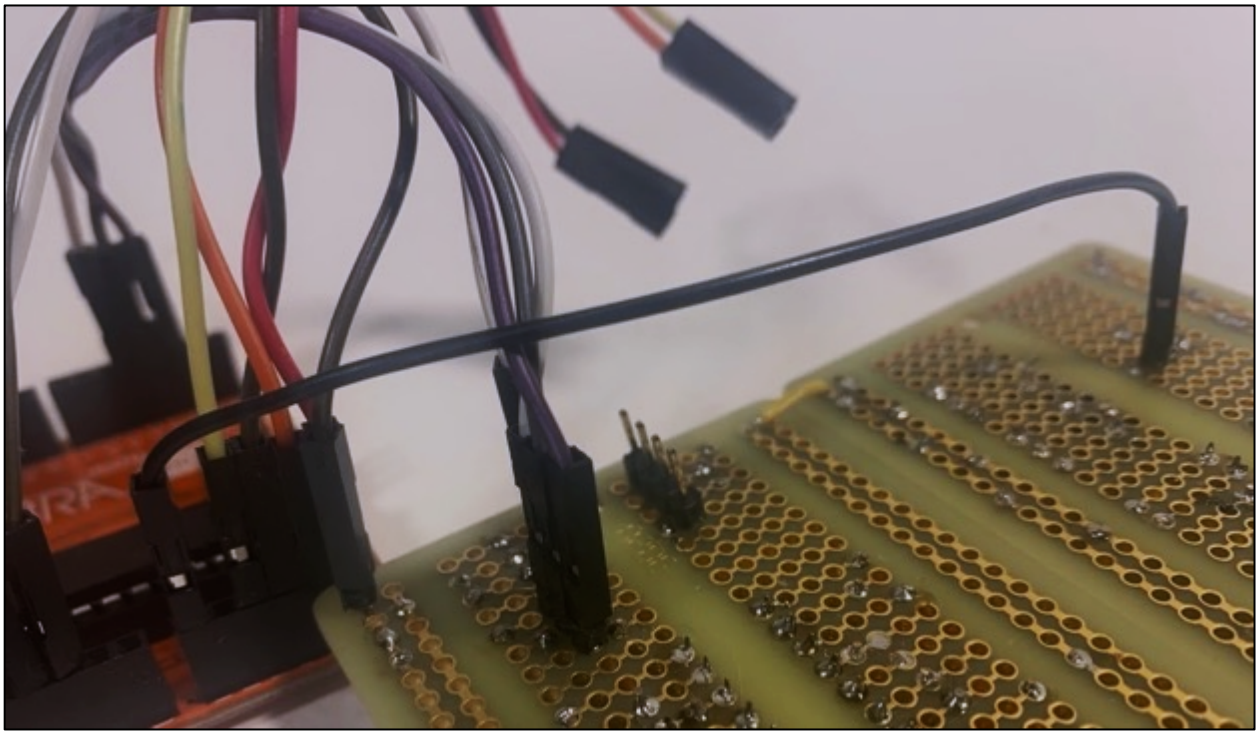

Unpainted Matrix Graphing Calculator



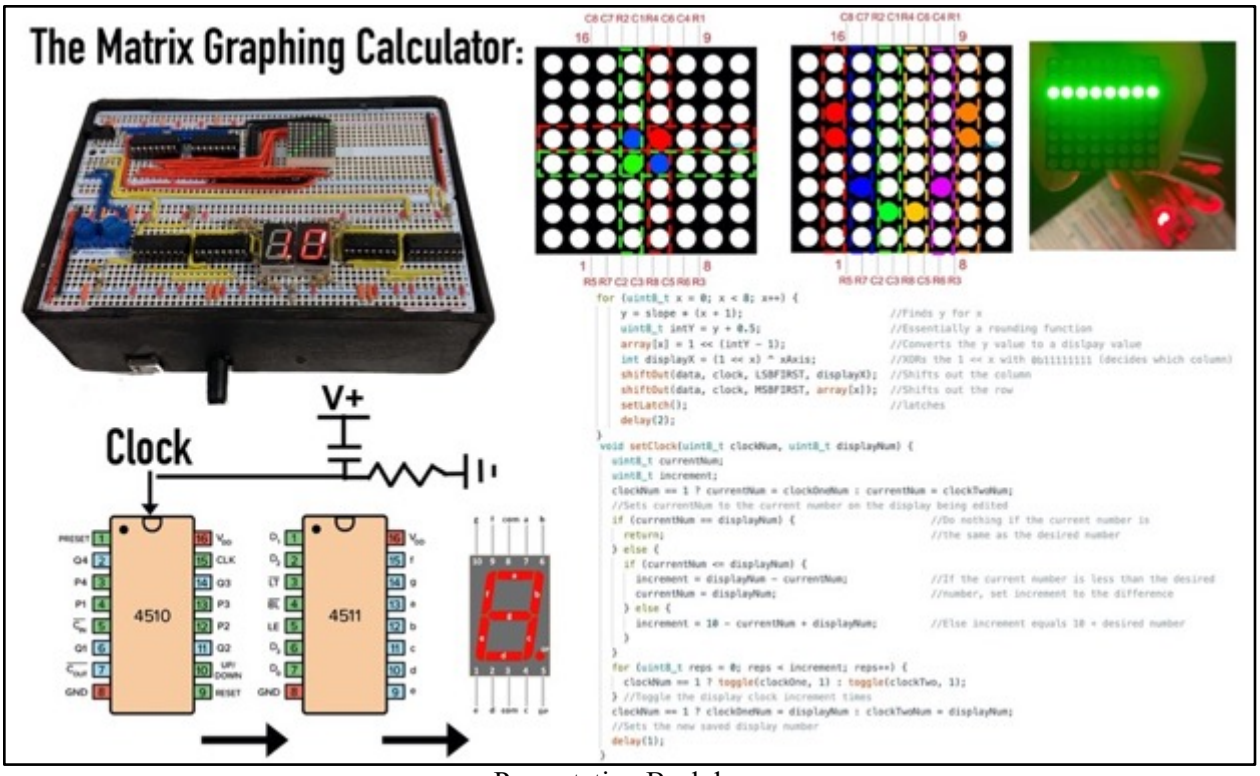
Testing Matrix Wiring and POV




Breadboard Prototype Without Nano



"Ports" on Back of Perma Proto Boards




Presentation Backdrop

## Code

```
// Project   : Short ISP: Matrix Graphing Calculator
// Purpose   : Graphs linear relations onto an 8x8 LED matrix.
// Course    : ICS3U
// Author    : Liam McCartney
// Date      : 2022 12 3
// MCU       : 328p
// Status    : Working
// Reference : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/ISPs.html#logs

// Variables :
#define data 2
#define latch 3  //Defining pins on the shift register
#define clock 4
#define duration 1

#define clockOne A3  //Defining 7-seg pins.
#define clockTwo A5

#define rise 16           //Rise and Run potentiometer pins
#define run 18

uint8_t clockOneNum = 0;  //Saved 7-seg display states
uint8_t clockTwoNum = 0;

uint32_t history[50]; //Saved rolling average

int array[8];                     //The row data for the matrix is stored here,
                                    essentially a bitmap
int xAxis = 0b11111111;           //For XOR later, ~ was causing problems
float y;                          //The use of floats here avoids some calculations
float x;
```

```
float slope = 0;                  //Has to be given an initial value for the first
                                    iteration
void setup() {
  Serial.begin(9600);
  while (!Serial);
  pinMode(data, OUTPUT);
  pinMode(latch, OUTPUT);  //Initializing shift register pins
  pinMode(clock, OUTPUT);
  clearDisplay(2);  //Clears the display from any previous programs

  pinMode(clockOne, OUTPUT);
  pinMode(clockTwo, OUTPUT);  //Pin Initialization

  digitalWrite(clockOne, 0);
  digitalWrite(clockTwo, 0);

  for (uint8_t i = 0; i < 50; i++) history[1] = 0; //populating history

  Serial.println("Hello World");
}

void loop() {
  float floatRise = analogRead(rise);     //Integers devided by integers do not
                                            return floats, and slope
  float floatRun = analogRead(run);       //has to be a float here.
  slope = floatRise / floatRun;           //Determining the slope
  smooth();                             //Calling the smooth function (running average)
  slope <= 1 ? underOne(slope) : overOne(slope); //Branching based on slope
}

void clearDisplay(uint8_t regNum) {                 //Sets shift register outputs to
0
  for (uint8_t reps = 0; reps < regNum; reps++)     //Clears regNum shift registers
    shiftOut(data, clock, MSBFIRST, 0b00000000);
}

void setLatch() {  //Turns the latch pin on and then off
  digitalWrite(latch, 1);
  digitalWrite(latch, 0);
}

void underOne(float slope) {
  //Solves for y using x
  for (uint8_t x = 0; x < 8; x++) {     //Iterates through 8 x's
    y = slope * (x + 1);                //Solving y
    uint8_t intY = y + 0.5;             //Rounding y to nearest int
    array[x] = 1 << (intY - 1);         //Setting array for row y to intended value
    int displayX = (1 << x) ^ xAxis;    //Setting ground outputs

    shiftOut(data, clock, LSBFIRST, displayX);  //Shift out ground first
    shiftOut(data, clock, MSBFIRST, array[x]);  //Shift out power (row y, column x)
    setLatch();                                 //Update display
  }
  uint8_t intSlope = slope;               //First digit of slope
  setClock(1, intSlope);                  //Sets 7-seg #1 to first digit
  intSlope = (slope - intSlope) * 10;     //Finds second digit of slope (decimal)
  setClock(2, intSlope);                  //Sets 7-seg #2 to 10ths digit
}

void overOne(float slope) {
  //Solves for x using y
  for (uint8_t y = 0; y < 8; y++) {     //Iterates through 8 y's
    x = (1 / slope) * (y + 1);          //Solving for x
```

```
    uint8_t intX = x - 0.5;             //Rounding x
    array[intX] = 1 << y;               //Setting array for row y to intended value
    int displayX = (1 << intX) ^ xAxis; //Setting ground outputs

    shiftOut(data, clock, LSBFIRST, displayX);  //Shift out ground first
    shiftOut(data, clock, MSBFIRST, array[intX]);//Shift out power (row y, column x)
    setLatch();                                 //Update display
  }
  uint8_t intSlope = slope;               //First digit of slope
  setClock(1, intSlope);                  //Sets 7-seg #1 to first digit
  intSlope = (slope - intSlope) * 10;     //Finds second digit of slope (decimal)
  setClock(2, intSlope);                  //Sets 7-seg #2 to 10ths digit
}

void setClock(uint8_t clockNum, uint8_t displayNum) {
  //clockNum tells the function which clock you want to wright to, so either clock 1
      or 2
  //displayNum is the number that we are going to display
  uint8_t currentNum;
  uint8_t increment;
  clockNum == 1 ? currentNum = clockOneNum : currentNum = clockTwoNum;
  //setting current number to the correct clock's saved value
  if (currentNum == displayNum) {
    return; //Do nothing (return) if the correct number is already being displayed
  } else {
    if (currentNum <= displayNum) {
      //If possible (displayNum > clockNum), toggle 'difference' times
      increment = displayNum - currentNum;
      currentNum = displayNum;
    } else {
      increment = 10 - currentNum + displayNum;
      //If the current clock number is greater, reset, then toggle displayNum times
    }
  }
  for (uint8_t reps = 0; reps < increment; reps++) {
    clockNum == 1 ? toggle(clockOne, 1) : toggle(clockTwo, 1);
    //toggle the correct number of times
  }
  clockNum == 1 ? clockOneNum = displayNum : clockTwoNum = displayNum;
  //update the current saved number
  delay(5); //the delay in the loop is technically in the setClock function
}

void toggle(uint8_t pin, uint8_t bit) {
  //toggles pin 'pin' from high to low or low to high depending on 'bit'
  if (bit == 1) {
    digitalWrite(pin, 1);
    digitalWrite(pin, 0);
  } else {
    digitalWrite(pin, 0);
    digitalWrite(pin, 1);
  }
}

void smooth() {
  //rolling array to smooth out problems
  for (uint16_t i = 0; i < 49; i++) history[i] = history[i + 1];
  history[49] = slope;  //Shifting indices
  float sum = 0;
  for (uint8_t i = 0; i < 50; i++) sum = sum + history[i];
  //Finding total of all past 50 slopes
  slope = sum / 50; //Average slope
}
```

## Reflection

In many ways, I believe that this report was a benchmark. Not only was this my first ever ISP, but my DER is now over 50% grade 11 content. As for how the project went, I am quite pleased. I will give a timeline of the whole thing:

I got working on it pretty early, I think I started figuring out the matrix the weekend following the Wednesday proposal submission. There were quite honestly not too many design hitches. Not to say that the progress was extremely fast, but it went smoothly without too much redesigning (except for re-wiring the matrix a few times). I think I had a working graph after about three weeks and finished the prototype three-four weeks before the presentations. There was a bit of a break around then to focus on other summative tasks, but by two weeks before the presentation, I had moved on to soldering. I have a soldering iron at home, so with one full day and a few hours here and there, the Perma Proto was ready for debugging a week before the presentations. I ended up needing that time, a few things did not go to plan with the soldering. Firstly, I used female headers for the first and for the last time. Only about half the pins of my matrix had connectivity to the board. In the end, I had to scrap the headers and pull them out. Secondly, I mis-soldered two wires and that took an embarrassingly long time to find, as well as one loose joint. Finally, I built the case in a few hours on the Saturday before the presentations. I opted for wood because I knew it was reliable. I had time to print the weekend before I built the case, but after seeing other classmates' cases go wrong in the printer, I decided to go with the reliable option instead and tackle 3D printing in the next ISP.

There are a few other things I want to reflect on. First off, the use of the 4510 and 4511 as opposed to chaining on more shift registers. I mentioned it briefly, but I had two things in mind when making that choice. Firstly, to have a one-pin method of controlling a seven-segment display, but it was also because I thought that the idea was interesting. I do not know if I would make the same choice again in the future, but I did learn a lot from it. The issue of the 4510 saving its count was a fun thing to solve, though somewhat frustrating because it is not on the datasheet anywhere (I did multiple tests and they all confirmed that the 4510 has some form of memory). Finally, I want to end off by talking about my video(s). I had the idea to use software named Manim. Manim is meant for illustrating math videos, and I used it to animate multiple graphs. I am proud of the result, especially since it was such a last-minute decision. I got suddenly inspired to use it on Friday. However, Manim is quite complex. To use it, I had to install lots of supporting software such as python, FFMPEG, and LaTex. It was quite an ordeal and took my whole Saturday to learn how to use it, resulting in this somewhat close-to-deadline submission. I hope to use it more in the future now that I have some knowledge of it and have seen the results.

# Project 2.3a Breadboard ATmega328p

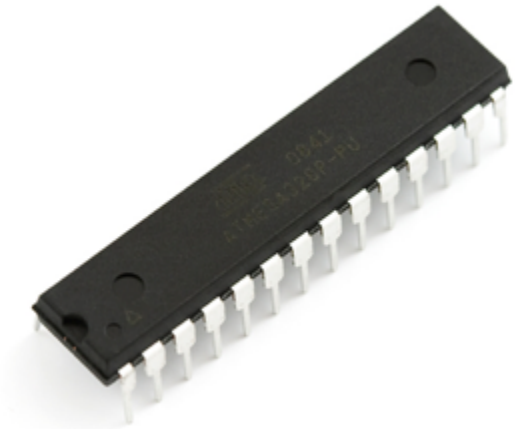
## Purpose

The purpose of the breadboard ATmega328p is to take a closer look at the ATmega328p without its supporting cast to better understand its functions and those of microcontrollers in general, as well as introduce new components such as quartz crystals and voltage regulators.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#BreadboardMega
Crystal Oscillator: https://en.wikipedia.org/wiki/Crystal_oscillator#Resonance_modes
Piezoelectricity: https://en.wikipedia.org/wiki/Piezoelectricity#Frequency_standard
Piezoelectricity - Steve Mould: https://www.youtube.com/watch?v=wcJXA8IqYl8&ab_channel=SteveMould
Quartz Crystals - Steve Mould: https://www.youtube.com/watch?v=_2By2ane2I4&ab_channel=SteveMould

## Procedure



Fundamentally, an ATmega328p in a breadboard and on an Arduino Uno/Nano is the same thing. In fact, the microcontroller used in this project could be replaced by one from any Uno. The only difference between this project and previous ones is that while the microcontroller itself might be the same, the supporting cast of the Arduinos is now gone. Seen on the right is an ATmega328p in a DIP package. To use it without an Arduino, some infrastructure is required. Most of this is simple, such as connecting the necessary communication pins to the chip from a computer, wiring up the reset pin, and attaching the chip to power and ground. However, there is one more important step, and that is adding a crystal oscillator to the circuit.

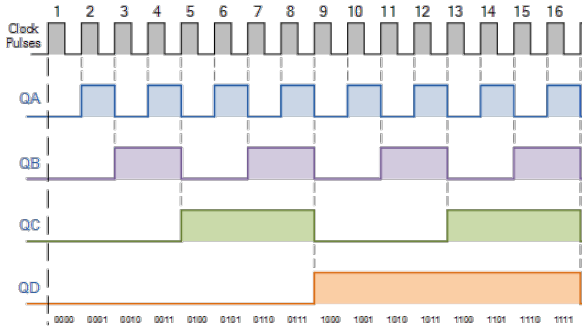
## Quartz Crystal Oscillators

Timekeeping is essential for any complex electronics. This idea has been mentioned multiple before, but it is more important now than ever. Circuits need to be reliable, and they need to be deterministic. Everything should take a set amount of time, and this time is measured in clock cycles. For example, when in past projects an Arduino executed `delay(1000)`, it would not actually wait one second. It would instead wait for 16,000,000 clock cycles, or the equivalent of one second (since Arduino Unos and Nanos run at 16MHz). 16,000,000 is obviously a very large and precise number, and due to the magnitude and precision, computers with clocks that fast cannot rely on timers that have already been discussed, such as the 555 (which utilizes resistor-capacitor pairs). Instead, devices with high clock speeds rely on crystal oscillators, one of the most accurate ways to measure time. While still not 100% accurate, crystal oscillators are more than adequate for the task of keeping time for a microcontroller; losses will only be a matter of seconds every year.

To achieve this high precision timing, two physics phenomena are exploited to ensure a constant, repeatable oscillation of a crystal tuning fork as shown to the right. The first is piezoelectricity. This is the phenomenon where certain materials produce a voltage difference when the material experiences mechanical stress. It also works in reverse, where a voltage will cause such a material to deform. To generate a clock cycle, a crystal tuning fork is exposed to a voltage, causing it to deform. This then causes a voltage in the tuning fork, which can be measured as a clock cycle. If the crystal is continuously exposed to a voltage, it will begin to oscillate at its resonant frequency, and this resonant frequency will be the frequency of the clock. Resonance, the phenomenon responsible for resonant frequencies, is the second phenomenon used in crystal oscillators. A resonant frequency is a frequency of oscillation at which the object oscillating will oscillate with a higher amplitude. This resonant frequency can be precisely altered to be almost exactly 16 MHz by either taking material off the crystal or by adding tiny bits of gold. Through this process, the crystal will emit a voltage with each oscillation, providing the computer with a 16 MHz clock, or any other needed frequency.

This of course is not the same as a watch, and having a reliable crystal oscillator does not keep time by itself. To keep real-time, or in other words, to use a crystal oscillator for a real-time clock (RTC), there are two main methods. Firstly, and most simply, there can be circuitry to count each clock cycle. This would be how the Arduino keeps time. When it is told to delay for 1000 milliseconds, it counts 16,000,000 clock cycles where it does nothing (more accurately counts to 16000, or 1 millisecond, a total of 1000 times).

The other method of keeping time is to specifically choose a frequency that is a power of two. The most common example of this would be a 32 kHz crystal, or more accurately exactly 32768 ($2^{15}$) Hz. With frequencies that are a power of two, we can use frequency division to count to precisely one second. There are many circuits capable of dividing frequencies by two in a cascade. In fact, this is how binary counting circuits work. If the 32 kHz frequency were divided by two fifteen times, the output frequency would be exactly 1 Hz. The diagram to the right shows what would be the first four phases of this division cascade. Both methods of counting have their benefits.

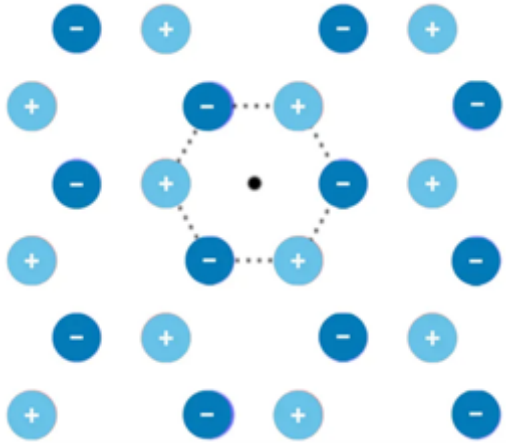



In general, counting each clock cycle with higher frequencies is more accurate than frequency division of slower clocks, and "loses" less time. This makes it the choice for applications such as computers. However, where exact precession is outweighed by cost and simplicity, 32 kHz RTCs are generally better options. Somewhat ironically, some common applications of the less accurate method are watches and stopwatches.

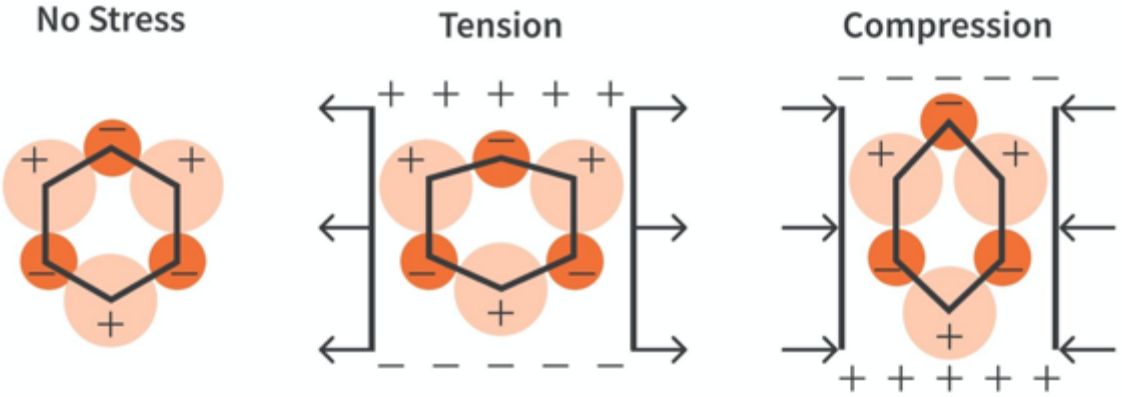
## Piezoelectricity

As a disclaimer, an understanding of piezoelectricity is not necessary to understand the significance of quartz oscillators. Piezoelectricity is arguably purely chemistry even though electricity is involved. However, it is still interesting.

To the right is a diagram of a quartz crystal lattice. While it appears to be a two-dimensional image, this is one of the most important properties of a piezoelectric material. They must be crystalline, and there must be some viewing angle where the three-dimensional lattice appears to be two-dimensional. This assures that what happens on a micro scale is the same as what happens on a macro scale. The next piece of criteria is that the crystal must contain polar bonds. This is why the silicon and oxygen are depicted with + and – signs; an oxygen-silicon bond is polar. Finally, a single section of the lattice must not contain a shape where a positive or negative atom is directly across from an atom of the same polarity.



This last point is important, because as shown below, when a piezoelectric material is deformed, it changes the physical shape of the entire lattice of the object. While there were always polar bonds, the crystal as a whole did not have a dipole moment. However, when the crystal is compressed or stretched, the latices change shape so that the individual atoms shift in such a way that the whole structure develops a positive and negative pole.

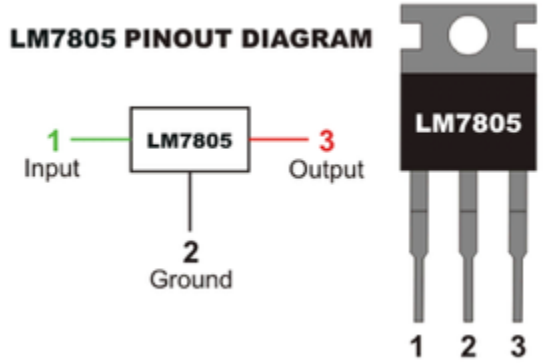



This is shown above, as the two atoms on the left and right only move horizontally, whereas the middle atoms move vertically. The horizontal atoms cancel out each other's movement, not changing the polarity of the structure, but the vertical movement of both the positive and negative central atoms creates a more negative and a more positive end of each structure, giving the whole lattice a dipole moment. Had opposite atoms been of the same polarity, the vertical vectors would have also cancelled, leading to a crystal with no dipole moment.

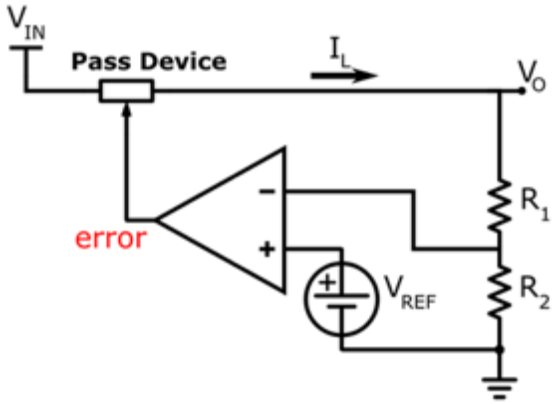
These two induced poles now provide a potential difference across the crystal, meaning a voltage is produced. This is the voltage that is used as a clock signal. To begin the oscillation, a voltage is applied to the crystal. The reason that the crystal deforms due to a voltage is the same as the previous explanation but in reverse. When a mechanical deformation causes a voltage, it is called direct piezoelectricity, and when a voltage causes a deformation, it is called inverse piezoelectricity.

## Voltage Regulation

While not strictly necessary, it is a good idea to have a voltage regulator on the breadboard to protect the ATmega from excessive voltage. As their name suggests, voltage regulators regulate voltage. More specifically, they can take in any voltage within a specified range and output a fixed voltage. Since the ATMega328p has a recommended operating voltage of 5 V, a 7805 5 V regulator is used. The 7805 is part of a series of regulators, labelled 78XX where XX is the output voltage. As seen to the right, the wiring on the 7805 is quite simple. However, it is not as easy as wiring up a power supply greater than 5 V. First, the input must be between 7 V and 35 V. If it is not, the regulator will overheat and trigger a shutdown to protect the pass transistor. As well as the input range, there must also be capacitors limiting the fluctuation of the input and output voltage. The capacitor on the output smooths the resulting 5 V, and the input capacitor prevents input spikes which would generate a great deal of heat that could possibly trigger a thermal shutdown.




The reason that heat is an issue with the 7805 is because the 7805 is a linear voltage regulator. This means that excess voltage is converted to heat. While this works well, is simple, and makes intuitive sense, it is not necessarily the most efficient. This method of shaving off voltage wastes energy. With a 12 V input, the L78S05 (2 A, 5 V regulator), over half the energy being inputted to the system can be wasted as heat. This heat is generated as the internal pass transistor blocks voltage from continuing through the circuit. It does this by comparing the current output voltage to a reference (desired) voltage, and depending on the output of the comparison, either allows or blocks the flow of electrons.

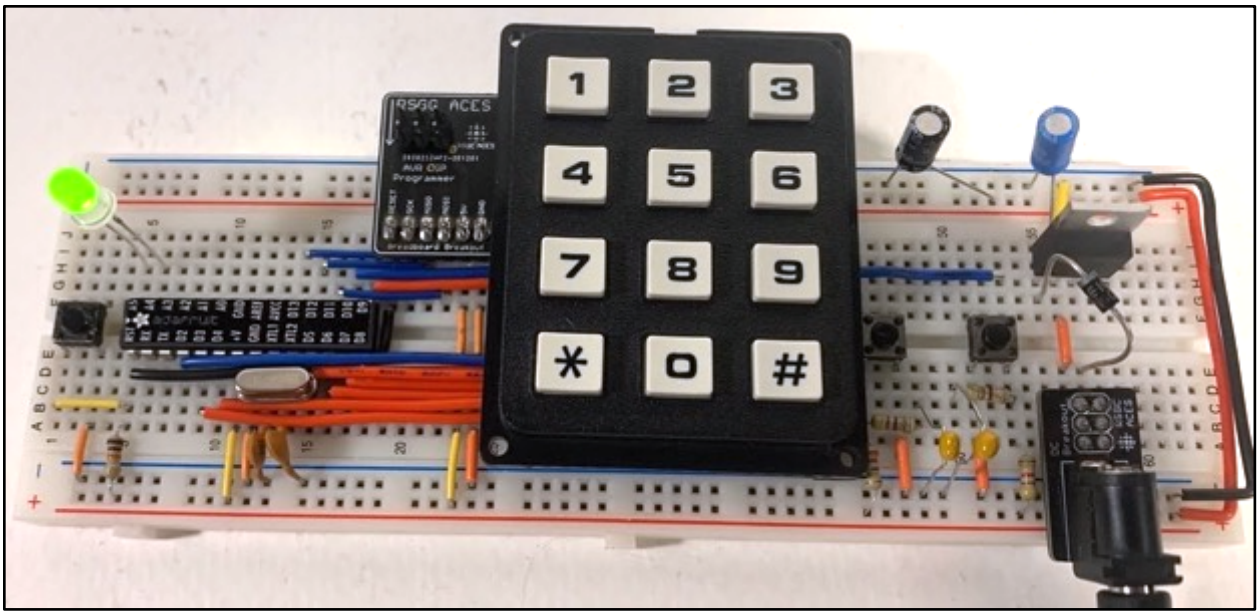



## Hardware & Software

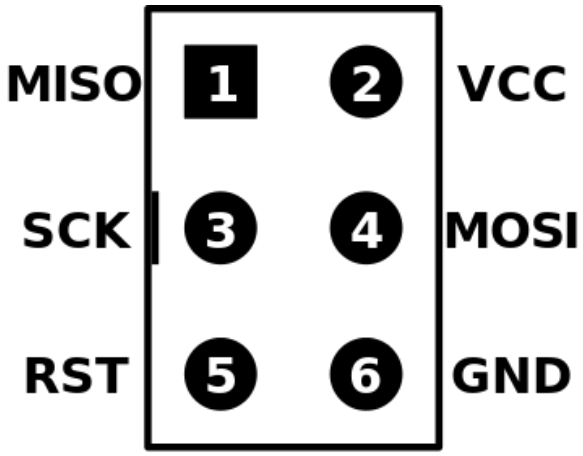
The build for this project is a simple "lock" circuit. It is a "lock" in quotations because all that happens is a bicolor LED turns red when the lock is locked, and green when unlocked. The device used a 12-key keypad and comes with a pre-set password that can only be changed in code. Once unlocked with the initial password, the user can then set the password to anything they want (excluding characters # and *). The length of the password is set in the code. To reset the password, the user first unlocks the device before entering the new password and then presses the save password button. To lock the device, the user presses the lock button (this is the button that uses an external interrupt on digital pin 2). Finally, there is a reset button attached to the ATmega itself that resets the whole system, voiding the saved password.

**Parts Table**

| Description | Quantity |
|---|---|
| ATmega328p | 1 |
| 16 MHz Crystal Oscillator | 1 |
| Push Button (Normally Open) | 3 |
| 0.1 µF Capacitor | 2 |
| 1 µF Capacitor | 1 |
| 10 µF Capacitor | 1 |
| 20 pF Capacitor | 2 |
| 12 Key Keypad | 1 |
| Bicolor LED (Red Green) | 1 |
| Power Diode | 1 |
| L7805 Voltage Regulator | 1 |
| 470 Ω Fixed Resistor | 4 |
| 1 kΩ Fixed Resistor | 1 |
| Jumper Wires | ** |




Above is a picture of the circuit in its unlocked state (shown by the green LED). The save password and lock button are to the right (both debounced), and the reset button is to the left. 9 V power is supplied to a 5 V voltage regulator, stepping down the voltage to the necessary levels for the ATmega. The crystal is in place with its two shoulder capacitors, and the keypad is wired to the microcontroller.

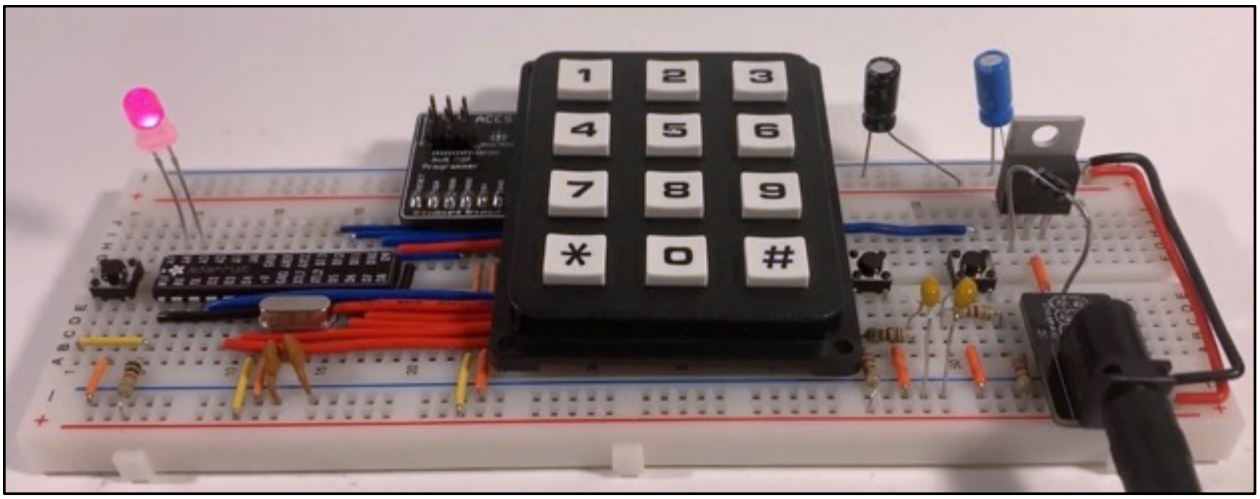



Seen to the left of the keypad, there is a PCB with a 2×3 set of input pins. This is a breakout board for the ISP pins coming from the AVR programmer. The PCB makes the 2×3 layout breadboard compatible (traditional ISP headers, as seen to the right, are not breadboard compatible). The ISP pins are MISO (master in slave out), MOSI (master out slave in), SCK (serial clock), reset, power, and ground. These pins are quite similar to SPI, except there is no slave select as ISP is meant for one-on-one communication. ISP stands for in-system programming and is a communication method meant for programming things such as microcontrollers. It relies on serial communication between the master and the slave, with one line for each direction of communication (the MISO and MOSI lines are meant for the different directions of communication; one is from the master to the slave, and one is from the slave to the master).

ISP communication is important for this project because it has some key differences from the usual USB to serial conversion. Normally when using USB to serial, the Arduino board and computer can communicate through the bootloader (which is why the bootloader has to be flashed prior to USB to serial programming). The bootloader is necessary for the serial monitor in the Arduino IDE because of the way that the computer is connected to the Arduino through a USB port. Since the end terminal of MISO communication is the USB port on the computer, any data meant for the computer must be readable by a USB. This is a problem with ISP communication from the microcontroller to the computer since ISP protocol cannot be read by the computer through the USB port. Therefore, the MISO communication line is essentially useless while using a microcontroller connected through a USB terminal unless the user puts in extra effort to surpass this issue, making debugging tools such as the serial monitor unavailable.

## Media

Project Video: https://youtu.be/0XS7ISmoUVg




Circuit in Locked State

## Code

```
// Project    : 2.3 Keypad Password
// Purpose    : A device capable of remembering a password and checking if it is
                correct
// Course     : ICS3U
// Author     : Liam McCartney
// Date       : 2023 01 12
// MCU        : 328p
// Status     : Working
// Reference  : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#BreadboardMega

#include "Keypad.h"

#define ROWS 4  //Keypad setup
#define COLS 3

volatile boolean unlocked = false;

char keys[ROWS][COLS] = {
  { '#', '0', '*' },
  { '9', '8', '7' },
  { '6', '5', '4' },
  { '3', '2', '1' }
};
uint8_t rowPins[ROWS] = { 6, 7, 8, 9 };  //connect the row pinouts of the keypad
uint8_t colPins[COLS] = { 3, 4, 5 };     //connect the column pinouts of the keypad
Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS);

#define length 6
uint8_t password[length] = { 8, 7, 8, 9 , 9, 9};
uint8_t history[length];
//The password is initially pre-set until the user choses one

#define ledRed A5
#define ledGreen A4         //Pin definitions
#define savePassword 10
#define resetPin 2

void setup() {
```

```
  pinMode(ledRed, OUTPUT);
  pinMode(ledGreen, OUTPUT);
  digitalWrite(ledRed, 1);
  digitalWrite(ledGreen, 0);
  //Making the bicolor LED red to show the device is "locked"

  attachInterrupt(digitalPinToInterrupt(resetPin), reset, FALLING);
  //This interrupt listens the reset pin and calls the reset function
}

void loop() {
  if (unlocked) {
    digitalWrite(ledGreen, 1);
    digitalWrite(ledRed, 0);
    //Set the LED to green to show that the device is unlocked

    char key = keypad.getKey();
    //Reads the current key pressed (even if it no key is pressed)

    if (key != NO_KEY) {
      //If the key pressed was actually a key... (not no key)

      for (int i = 0; i < length; i++) {
        history[i] = history[i + 1];
      } //Shift ever index in history back 1

      history[length - 1] = key - 48;
      //Add the current key as the last index
    }

    if(!digitalRead(savePassword)) {
      //If the save password button IS pressed (on by default, PBNC)
      for(int i = 0; i < length; i ++) password[i] = history[i];
      //Make the password the current history
    }
  } else {
    //If not unlocked
    digitalWrite(ledGreen, 0);
    digitalWrite(ledRed, 1);
    //Make the LED red

    char key = keypad.getKey();
    //Get the current key

    if (key != NO_KEY) {
      //If it is a real key...
      for (int i = 0; i < length; i++) {
        history[i] = history[i + 1];
      }
      history[length - 1] = key - 48;
      //Again shifting history by 1 index then adding current read
      unlocked = true; //No matter what, the lock unlocks here

      for (int i = 0; i < length; i++) {
        if (history[i] != password[i]) unlocked = false;
      }
      //But it only stays unlocked if it "survives" this for loop
    }
  }
}
void reset() {
  unlocked = false;
}
```

## Reflection

This report was definitely an interesting one. It is the first of the winter season, which may seem irrelevant, but I now have to work both days of the weekend. This significantly cuts down on the time that I can commit to school. I imagine that it will be a struggle some weekends to just finish normal homework, let alone spend hours on a single assignment. However, the thing about hardware is that DERs are not a surprise, and through some good planning and time management, I think I should make it through relatively smoothly. This week I was caught off-guard, I was not expecting the first week back to come with so much work in all my courses. In the future, with the knowledge of how this submission went, I will definitely be starting as early as I possibly can to alleviate stress, and hopefully preserve some sleep. Asides from time management, I think this report went well. I tried something new with a full page of essentially pure chemistry, and I am not sure about it. On one hand, I found it interesting and it represents learning that I did and I now want to write about it, plus it is directly related to crystal oscillators which are part of the course material. On the other hand, the chemistry of piezoelectricity itself would probably not be considered part of the hardware course material, and in the end, this is a hardware report. Either way, I enjoyed writing that section and I will leave it as it is for now.

# Project 2.3b Perma-Proto ATmega328p

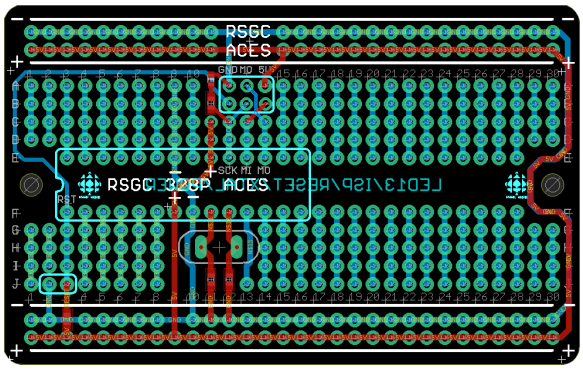
## Purpose

The purpose of the Perma-proto ATmega328p is to move towards the end goal of a fully permanent version of the circuit from the previous project. In addition, this project provides inspiration for the final step (2.3c) in the form of a custom half-size perma-proto board built specifically for this project. In this project, the ATmega328p is moved to a fully permanent board with a case.

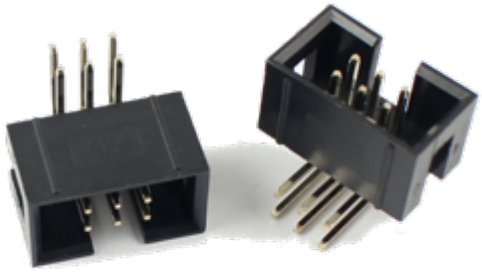
## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#PermaProtoMega

## Procedure



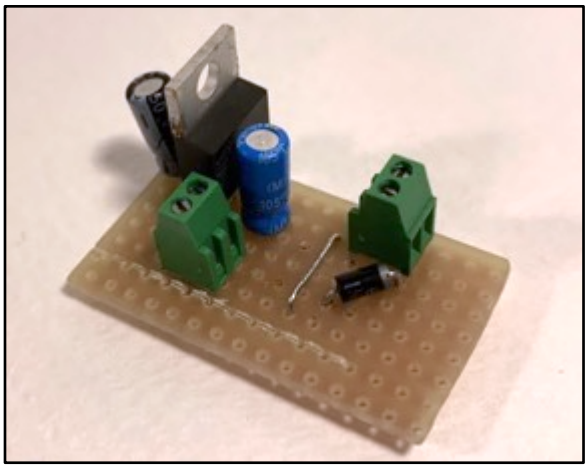

2.3b begins with a great example of how PCBs can be used to simplify builds and make them look neat. To the right is an image of the half-sized board used in this project. It is a customized version of Adafruit's original perma-proto. This version of the board includes traces that limit the amount of exposed wiring by hiding it inside the substrate (the body of the PCB). As shown, the board includes features such as pre-connected power and ground rails on each side, pre-installed surface mount shoulder capacitors for the crystal, power and ground connections for the appropriate pins on the ATmega, and most importantly broken-out ISP pins. This final point saves a great deal of board space and makes the overall product look much cleaner.



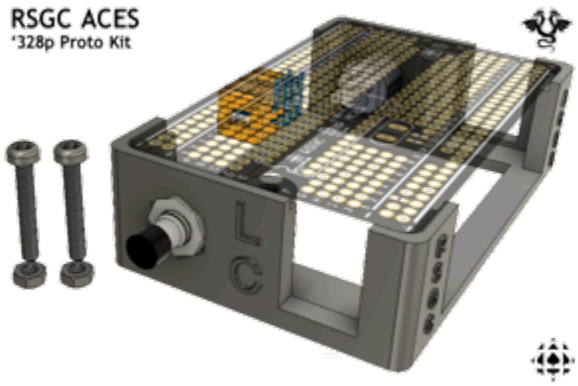
This new method of breaking out the ISP pins allows for a shrouded six-pin ISP header to be mounted underneath the 2×3 area and allows for ISP connections that cannot be attached incorrectly. The previous custom breakout board allowed for both the board itself to be placed on the wrong pins, as well as for the ISP cable to be attached to the breakout board backwards. This design stops both of those issues, with a small tab only allowing the ISP header to be inserted with the correct orientation.

As mentioned, one of the benefits of the shrouded ISP header is the board space that it saves. Board space is precious in this project. Half-size perma-proto boards do not have a lot of real estate to begin with, and with 16 out of 30 rows being occupied by the ATmega, the remaining space will be needed for the final step of PCB construction in 2.3c. Therefore, additional circuits such as voltage regulation must be off-board. In this case, a strip-board voltage regulator circuit is mounted underneath the perma-proto board and is connected through terminal blocks. A barrel jack is installed in the case through a custom hole.

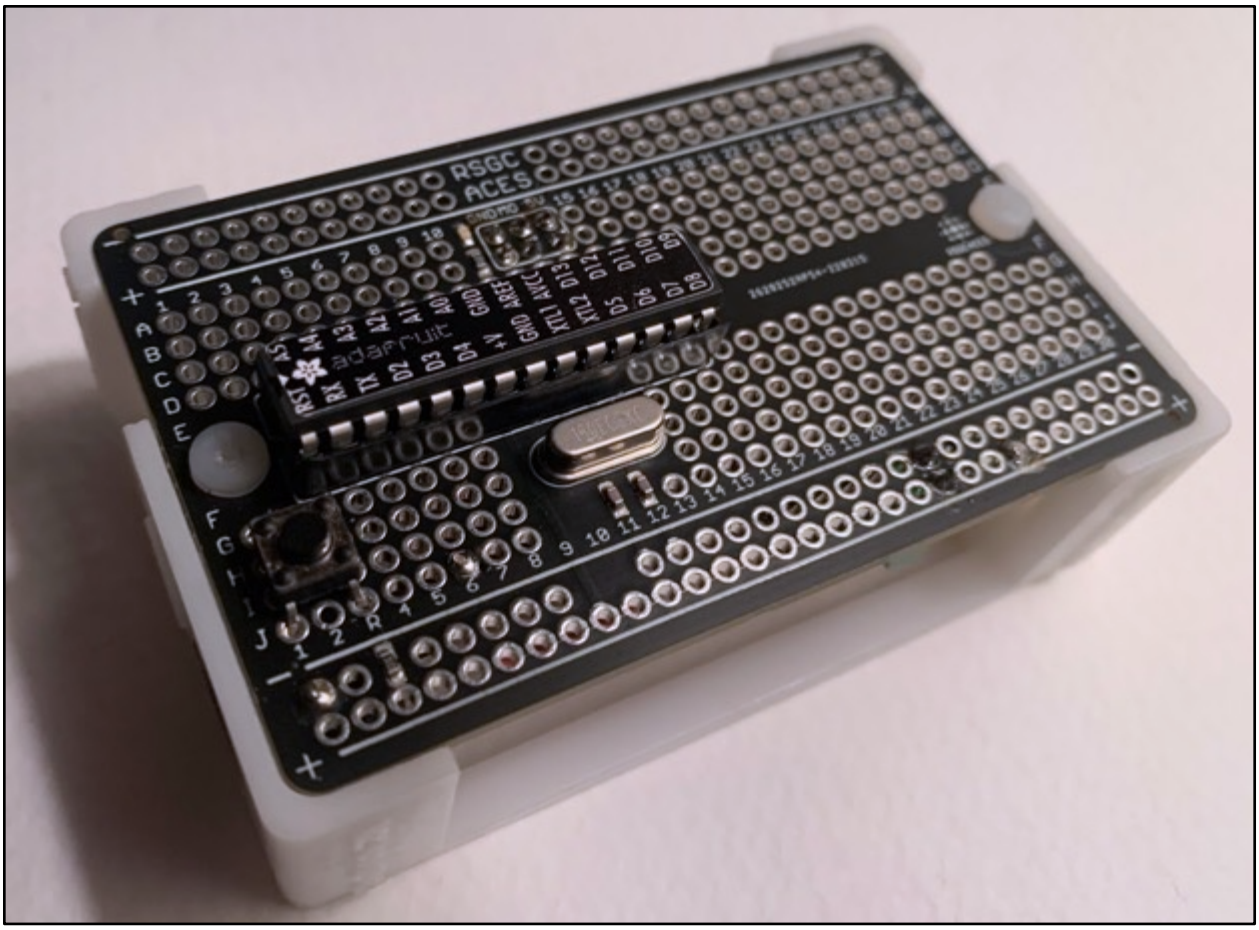
To the right is the voltage regulation circuit on its strip-board. Supply voltage comes in through the terminal on the left and comes out as 5 V on the right terminal. A simple permanent circuit like this has proven to be very useful so far, and will most likely be used in other projects as well. Therefore, a possible contestant for a PCB version of this circuit was designed in Eagle to compliment the coming lock PCB, as well as to be available for other projects in the future. This will eliminate some of the pains of strip boards, such as "runny" solder (due to the long strips of copper) and the fact that strip-board is hard to cut into shape.



As well as a custom perma-proto board, this project also comes with a case. Unlike previous cases, this case is a resin print. Resin prints are quite interesting and do not work like conventional 3D printing; they are made out of a resin that is exposed to UV light, which cures and solidifies the print. Resin printing has some advantages over thermal plastics, such as being smoother and sharper, but there are also some disadvantages. In this case, the biggest disadvantage is that resin does not melt due to heat, meaning that heat sets cannot be inserted for screws. Instead, the perma-proto is clamped onto the case with a screw and nut from either side.



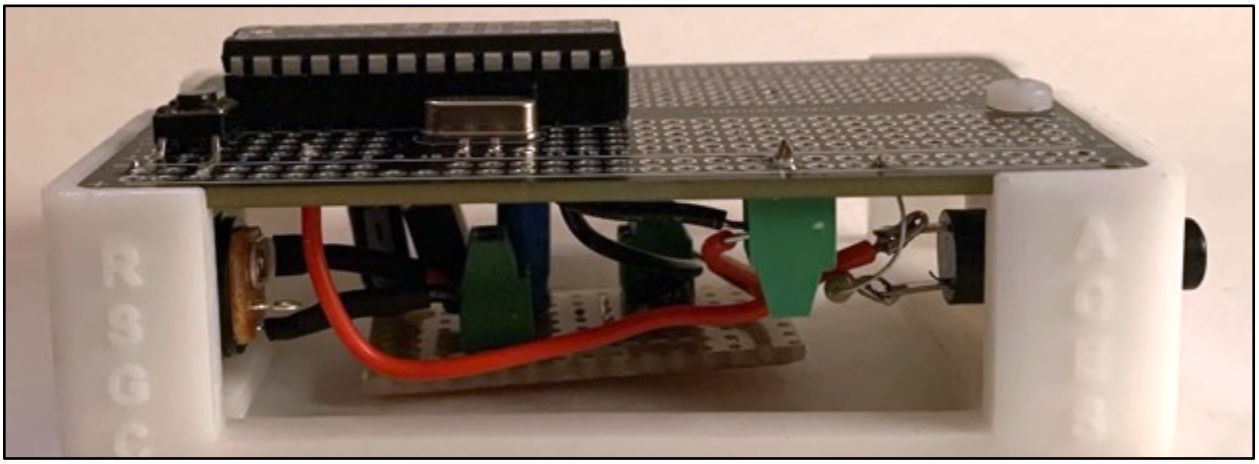

| Parts Table | |
|---|---|
| Description | Quantity |
| **Perma-Proto Board:** | N/A |
| RSGC ACES Perma-Proto 1/2 Sized PCB | 1 |
| ATmega328p Microcontroller | 1 |
| 16 MHz Quartz Crystal Oscillator | 1 |
| 28-Pin Kinked IC Socket | 1 |
| 2 × 3 Shrouded ISP Header (Wurth) | 1 |
| RSGC ACES 1/2 Size Perma Proto Mount | 1 |
| 2.1mm×5.5mm Schurter Power Jack | 1 |
| M3 5 mm Nylon Screws | 2 |
| Coloured PBNO with Leads | 1 |
| **Voltage Regulator:** | N/A |
| Strip Board (Custom Cut) | 1 |
| Power Diode | 1 |
| 1 µF Capacitor | 1 |
| 10 µF Capacitor | 1 |
| L7805 Voltage Regulator | 1 |
| Block Terminal | 3 |
| Heat Shrink Tubing | ** |
| Jumper Wire | ** |



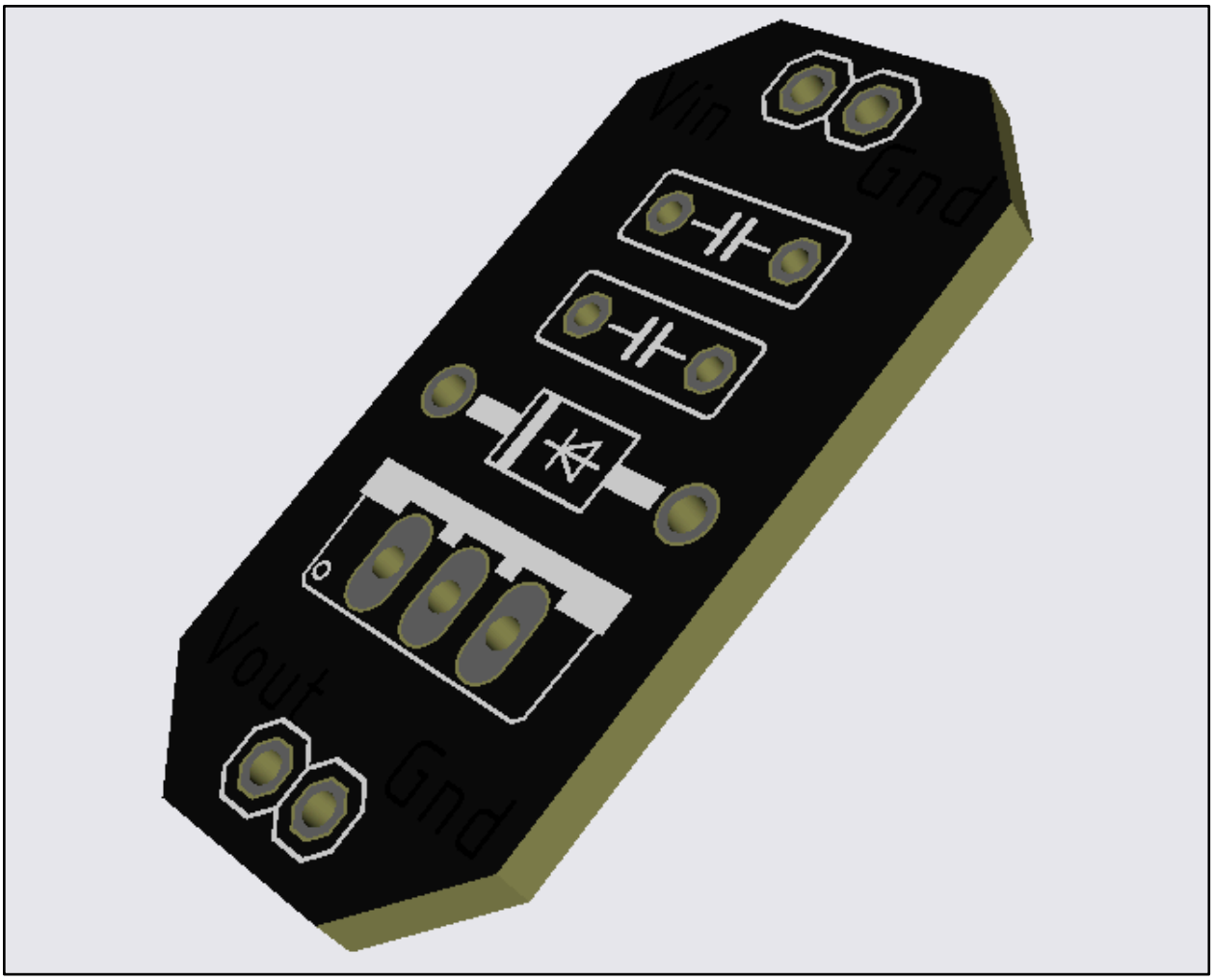

Above is a picture of the final assembled product. Hidden underneath the perma-proto is the voltage regulator and a button with a pull-up resistor attached to interrupt pin 0. In this case, there is no important use for the interrupt service routine (ISR) but the future build will make use of this button. In the sketch in this project, the ISR increments the duty cycle of the flashing LED on the SCK pin (digital pin 13) by 10% with each press to show that it functions.

## Media

Project Video: https://youtu.be/P-YspC0VpAA



Top and Underside of Perma-Proto




Possible PCB Design for Voltage Regulation (Designed in Eagle, Rendered in JLPCB)

## Reflection

It is hard to deny; project was quite simple. Funnily enough, I found the simplicity of the project very challenging, as it made it hard to find things to write about. This report is four pages, tied for my shortest ever. I have gotten used to writing a lot about topics in a project that I find fascinating, and here I did not find anything that I did/researched worthy of writing lots about. Not to say that there is nothing interesting, and also not that there is nothing worthy of writing lots about (for example, I believe one person rewrote the whole blink sketch to explore power usage and putting the microcontroller to sleep), but I instead focused on things that did not translate to words. Take, for example, the voltage regulation circuit. I spent at least an hour planning, soldering, and implementing it, and then also spent a few more hours learning the basics of Eagle to design a PCB to eventually replace the strip-board. However, I did not feel that these areas in which I invested time were that relevant to the project, and therefore I did not mention them at great length (especially because PCBs and PCB design are the focuses of the next report).

## Code

```
// Project   : Blink++ (Perma-Proto ATmega328p)
// Purpose   : Places a square wave on a digital pin with varying duty cycle
// Course    : ICS3U
// Author    : Liam McCartney
// Date      : 2023 01 21
// MCU       : 328P
// Status    : Working
// Reference : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#PermaProtoMega

#define interruptPin 2
#define duration 300
//Definitions

float timeOn;
float timeOff;
//These values cause problems when they are integers

volatile uint8_t count = 0;
//Used in the interrupt

void setup() {
  pinMode(SCK, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(interruptPin), increment, FALLING);
  //Call the increment function when the
}

void loop() {
  digitalWrite(SCK, 1);

  timeOn = duration * 0.01 * count;
  //0.01 * count turns count into a percentage to get the % duty cycle of duration
  delay(timeOn);

  digitalWrite(SCK, 0);

  timeOff = duration - timeOn;
  //The LED should be off for the remainder of duration
  delay(timeOff);
}

void increment() {
  while (!digitalRead(interruptPin)) digitalWrite(SCK, 1);
  //Keep the LED on while the interrupt is button is pressed
  count < 100 ? count += 10 : count = 0;
  //Increment the count by 10 unless it is 100 already
}
```

# Project 2.4 I2C Data Logger

## Theory

The Inter-Integrated Circuit. (I2C) Data Logger project is a project designed to introduce the task of creating an integrated system. In the context of the Data Logger, an integrated system refers to a device that can be left alone for an extended period of time and will perform some task. The device can then be used to gather information of the time period that it was left for. In the case of the Data Logger, the device gathers temperature information over a one-day period and displays the current temperature as well as storing a sample in electronically-erasable, programmable, read-only memory (EEPROM) every two seconds.

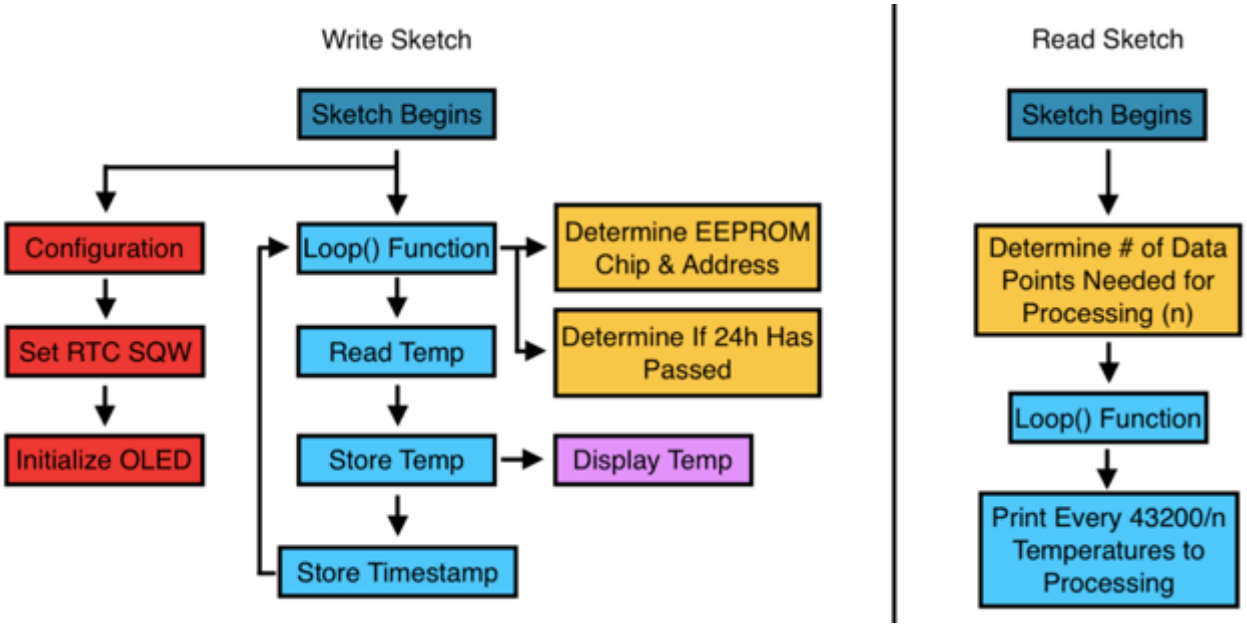
In addition to its function, the Data Logger also requires exploration of design in the form of a 3D-printed case and a fully soldered Perma-Proto board with the Data Logger circuit. To visualise the data, the projects also encourages a dive into Processing, an application that can be used for data analysis.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#datalogger
I2C Explanation: https://learn.sparkfun.com/tutorials/i2c/all

## Procedure

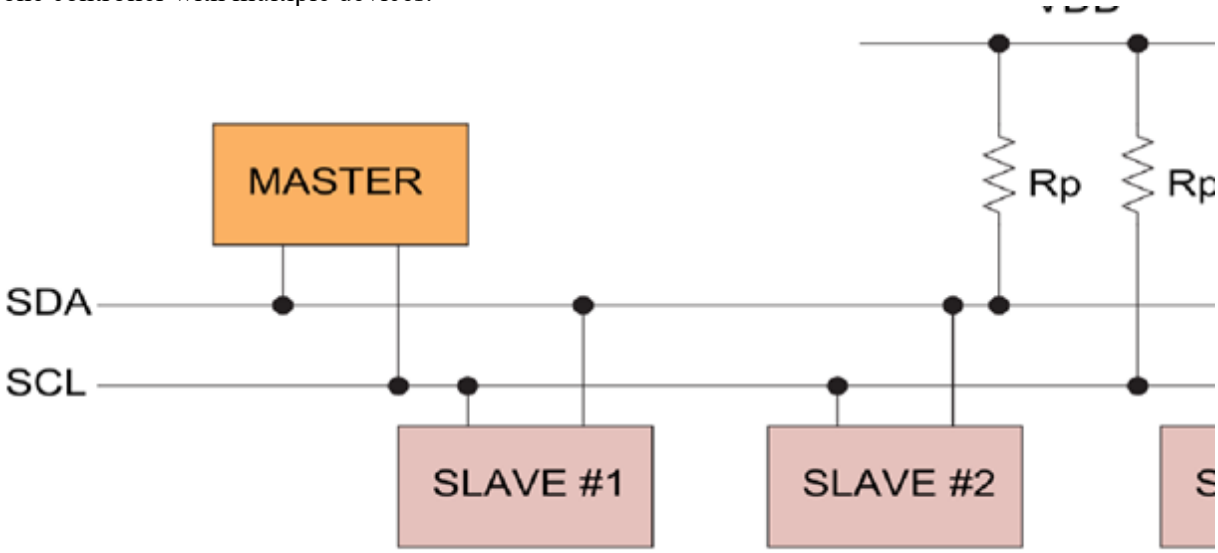
Before analysing the components of the Data Logger circuit, it is important to break down and clearly outline what the circuit has to do. With an integrated circuit such as this, it is easy to overlook parts of the process. Below is a flow diagram of the two sketches used by the circuit:




With this diagram we can create a checklist. If the circuit is capable of all of these things and the code has the needed functions, it will not be too difficult to piece them together to create a functioning project. The checklist is as follows: We must be able to 1. Set RTC square-wave timing 2. Initialize the OLED 3. Be able to read the temperature 4. Be able to store the temperature and timestamp (write to EEPROM) 5. Display the temperature on the OLED display 6. Print temperatures (read EEPROM).

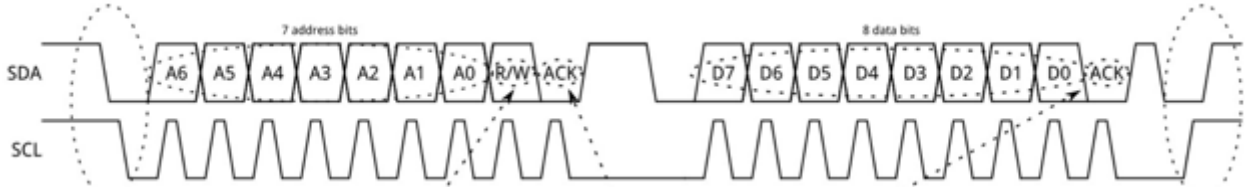
## I2C

While it is possible to skim over the workings of I2C data transfer, a deep look offers a better appreciation for the final result of this project. In I2C, there are only two communication lines, serial data (SDA) and serial clock (SCL). This is less lines than SPI and most other popular low-speed communication methods. While I2C can support multiple masters (controllers) and slaves (devices), in this situation there will only be one controller with multiple devices.




The first thing that it important to understand about I2C is the role of the two pull-up resistors on both the communication lines. These resistors are present because I2C devices use an open-drain configuration. This means that devices do not have the ability to set the communication lines high, they can only pull them low. This is important, as if the devices instead did have the ability to set a line high, there could be issues. Imagine a scenario where it is up to a single controller or device to keep the SDA high: no other devices would be able to change the state of the SDA line. With an open drain configuration, any controller or device can pull the SDA line low so long as it is not in use.

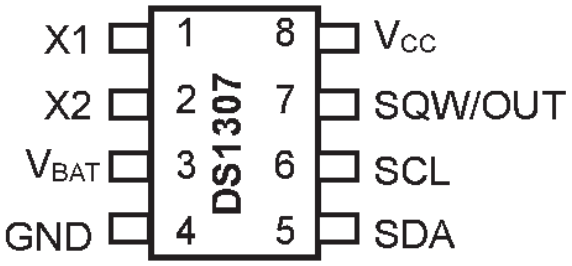
Finally, before looking at the timing, one last thing is important. I2C is a controller-speak-first communication method; there is no scenario where a device prompts the controller to read or write anything. To start communications, the controller pulls the SDA line low while the SCL (which the controller always controls) line is still high. This "wakes up" all I2C devices on the bus. Next, the controller writes the 7-bit address of the device it wishes to talk to (All I2C devices have specific addressed).




As well as the address, the controller sends a read/write bit where 1 is a read request and 0 is a write request. After this first byte, the device should send an acknowledgement (ACK) where it pulls SDA low after the controller releases. After the ACK, the controller can then send a byte of data to the device, where it will then again wait for an ACK. Once the controller is done with the transmission, it releases SDA after SCL, indicating the end of transmission.

## Hardware

In this project four main I2C components are used. These are the DS1307 RTC, the TC74 temperature sensor, the 24 LC256 EEPROM chips, and an SSD1306 OLED display. Beginning with the DS1307, a pinout can be seen to the right. The DS1307 is an RTC, and in this circuit it is used to tell the time when the circuit is turned on. The DS1307 has a 32.768 kHz crystal wired to its X1 and X2 pins to keep time. Inside the DS1307, there are seven important registers (not including 56 bytes of RAM).




| ADDRESS | BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 | FUNCTION | RANGE |
|---|---|---|---|---|---|---|---|---|---|---|
| 00h | CH | 10 Seconds | | | Seconds | | | | Seconds | 00–59 |
| 01h | 0 | 10 Minutes | | | Minutes | | | | Minutes | 00–59 |
| 02h | 0 | 12 | 10 Hour | 10 Hour | Hours | | | | Hours | 1–12 +AM/PM 00–23 |
| | | 24 | PM/ AM | | | | | | | |
| 03h | 0 | 0 | 0 | 0 | 0 | DAY | | | Day | 01–07 |
| 04h | 0 | 0 | 10 Date | | Date | | | | Date | 01–31 |
| 05h | 0 | 0 | 0 | 10 Month | Month | | | | Month | 01–12 |
| 06h | 10 Year | | | | Year | | | | Year | 00–99 |
| 07h | OUT | 0 | 0 | SQWE | 0 | 0 | RS1 | RS0 | Control | — |
| 08h–3Fh | | | | | | | | | RAM 56 x 8 | 00h–FFh |

Above is a diagram of the registers in the RTC. Besides the fact that the registers are formatted in BCD for whatever reason, the first six are relatively straight forward. It is the seventh register that it the most interesting. Looking back at the pinout, pin seven is named SQW. This is the square-wave pin that is built into the DS1307, and register seven is responsible for its configuration. This pin becomes important in the software portion of this project.

To format the square-wave, you first need to look at the table that represents the register. As seen below, only four bits in the register are important. Bit 7 represents what can be thought of as the idle state of the SQW pin. In this case its value does not matter. Bit 4 either turns on or off the square-wave. Finally, bits 0 and 1 represent a binary count as to the current output frequency.

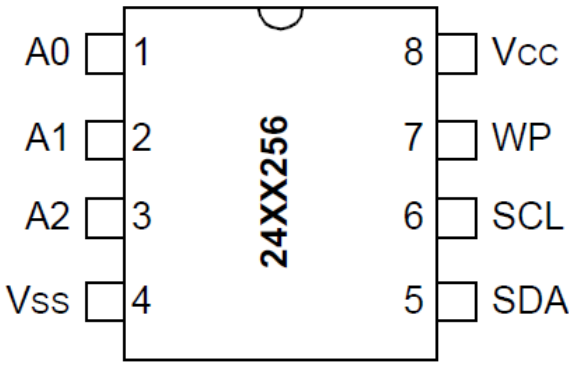
| BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|
| OUT | 0 | 0 | SQWE | 0 | 0 | RS1 | RS0 |

As seen below, each number made by bit 0 and 1 correspond to a different SQW speed. In this case we want a 1 Hz frequency, so using I2C communication we set register seven in the DS1307 to be 0b1001000 or 0x90 (144 decimal).

| RS1 | RS0 | SQW/OUT OUTPUT | SQWE | OUT |
|---|---|---|---|---|
| 0 | 0 | 1Hz | 1 | X |
| 0 | 1 | 4.096kHz | 1 | X |
| 1 | 0 | 8.192kHz | 1 | X |
| 1 | 1 | 32.768kHz | 1 | X |

Next is the 24LC256 EEPROM. Confusingly, sources do not seem to agree on the memory size of this IC. Some claim that this chip has a storage size of $2^{18}$ bits (262,144 bits), which is not very close to 256,000 bits (as in the name) at all. In fact, this is a 2.5% error in accuracy.

Either way, the 24 LC256 EEPROM chip is an I2C friendly eight-pin EEPROM IC. In this project, five of these ICs are used. With most I2C devices this would be a problem as they would all have the same address. However, the 24 LC 256 has a customisable address through the A0, A1, and A2 pins. By changing the binary value of these pins, the address can be changed from its base of hex 50. This allows for eight EEPROM chips to be used, granting 2,048,000 bits – 2,097,152 bits of storage (depending on the source you believe).

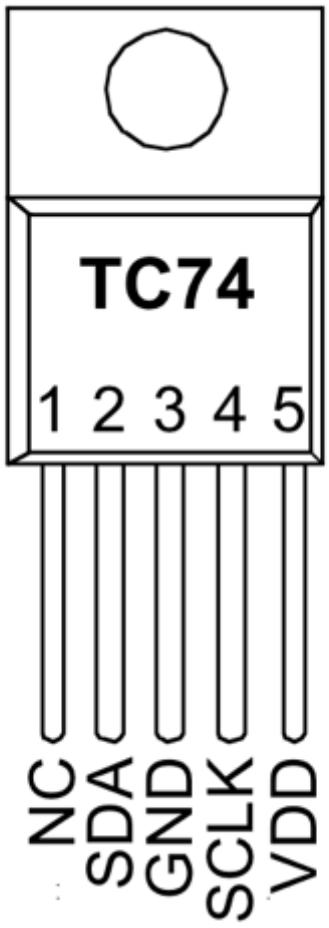



While an in depth converstation of how EEPROM works is not needed to understand the use of this IC, a good understanding of what the chip does, and EEPROM in general, is essential. As previously mentioned, EEPROM means electronically-erasable, programable, read-only memory. This essentially means non-volatile memory that you can edit. This is important for uses such as the Data Logger, where the device could be unplugged before the contents is read, and is the main benefit of using EEPROM: it stores data without power to the chip.

The next component is the TC74 temperature sensor. Interestingly, the TC74 has five pins, but only four are used, which is the most common number seen by I2C devices.

The TC74 has relatively fiew configurations compared to the DS1307, but some things can still be customized. Inside the TC74 are two registers, the first of which holds the temperature and can be read from, and the second which is the configuration register. In the configuration register there are only two important bits, and one of the two is read-only and says wether or not data is ready. The only bit that can be changed is the STANDBY switch, which controls the power level of the sensor. Writing a 0 to this bit puts the sensor to sleep where it draws almost no current and no longer takes temperature readings. Writing this bit back to a 1 re-enables the temperature readings.

The temperature range of the TC74 is -40 to 125 degrees Celcius. Since the range includes temperatures below 0 degrees, the first bit in the temperature register referes to the sign of the temperature while the next seven bits are the magnitude of the temperature.



The final I2C component used it the SSD1306 OLED display. This display is essentially a giant array of 128 × 32 LEDs. The most noteworthy thing about this device is how primitively it is controlled. This device requires every pixel to be assigned its own value from top left corner to the top right corner, then down one row and repeat. This leads to massive arrays of big numbers, using up lots of memory for each frame. In addition, because of the scrolling-refresh nature of the data stored, it is hard to manipulate multiple characters together. Like the TC74, the OLED display only has four pins to control its entire display.

## Software

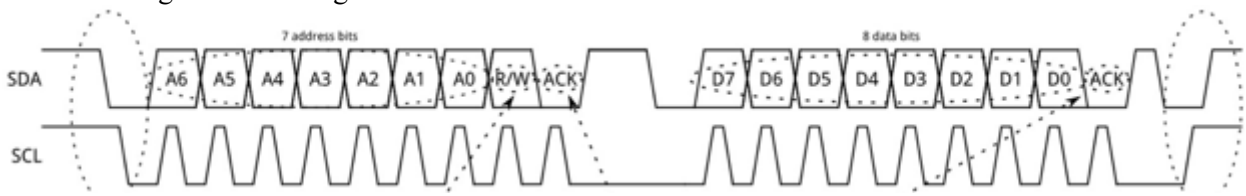
The most challenging piece of software in this project is by far using the SSD1306 without any libraries. The task with the OLED is to find a way to write any two-digit number, and to do that every digit has to be stored and accessible. The sheer size of the arrays needed to store any significant number of values/characters is enough to quickly overrun the storage capacity of the ATmega328p. However, there is a work-around to this error. By declaring variables locally in their own function, the global variable limit is never reached, and since only one variable is only ever loaded at once, neither is the local limit.

However, this is not a perfect solution. This then gives 10 functions that can each only be used with a specific number. To use them, one may be tempted to write an if ladder to go through every possible function. However, there is one interesting way to work around this issue as well:

```
void (*setDigitFunctions[10])(int) = {setZero, setOne, setTwo,
setThree, setFour, setFive, setSix, setSeven, setEight, setNine};
```

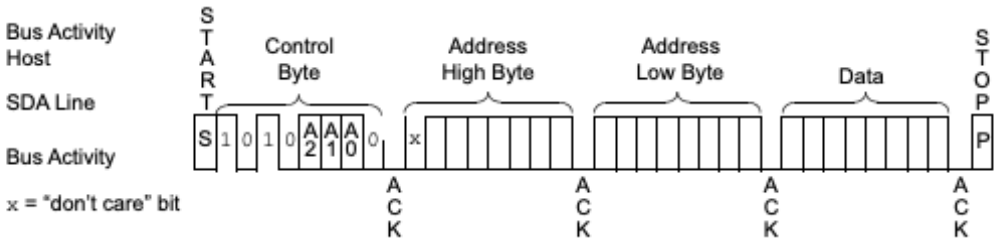
This function introduces the idea of pointers (denoted by the *). Pointers are essentially placeholder values that are simply the address in memory of the real destination. They can be thought of as a redirection towards the final destination. Here, `setZero` through `setNine` are functions that are declared elsewhere in the code which set a single global array to equal a saved character image. This function avoids having to call a function based on if statements by instead going straight to the function by making each function an index in an array. If we, for example, wanted to set the display to integer `n`, could call `setDigitFunctions[n]` to load the desired value into our display array.

Next is writing to and reading from the EEPROM ICs. This is transferable to the other I2C devices.




To begin with, the built in Arduino Wire library is used. This is essentially a library that handles the tricky parts of I2C for the user. This mainly includes starting and ending transmissions, as well as ACKs. The Arduino IDE comes with the built in functions `Wire.beginTransmission()`, `Wire.endTransmission()`, `Wire.write()`, and `Wire.requestFrom()`. The functions, in order, send a byte containing the device address and a 0 (write bit), release SDA while SCL is high, send a byte and wait for an ACK, and sends a byte containing the device address and a 1 (read bit).

We can now use those functions in conjunction with the EEPROM IC datasheet to create a function to write to EEPROM. First, we use `Wire.beginTransmission()` to begin a write request with the EEPROM. Then, as detailed in the datasheet, we send the high byte of the address in memory that we want to edit, followed by the low byte using `Wire.write()`. After that, we again use `Wire.write()` to send the device the desired value for the memory address. Finally, we end transmission with the device with `Wire.endTransmission()`.



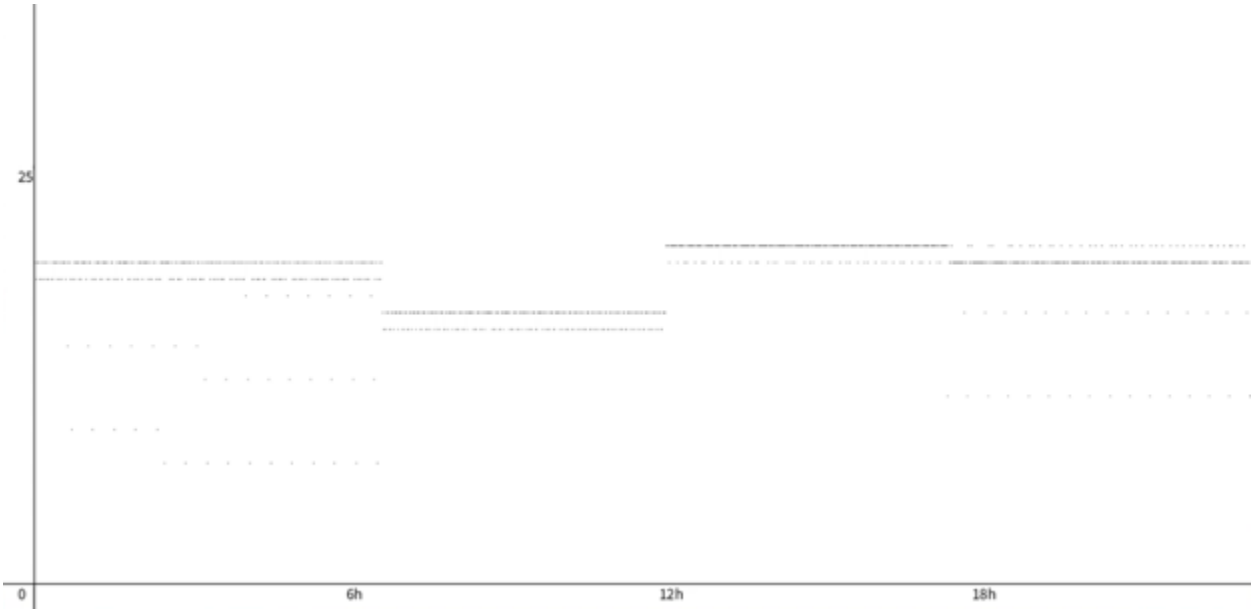

To the right is the final code for the `eepromWrite()` function. The function takes in the address of the specific EEPROM being written to, the address in memory, and the data for that address. With the use of wire, the I2C communication is greatly simplified. The delay at the end of the function is added to let the EEPROM finish its write cycle as it does not accept and communication while executing a write, so anything said to the chip while writing will be forever forgotten and will not be executed. Most sources recommend a delay of at least five milliseconds as seen to the right.

```
void eepromWrite(uint8_t chipAddress,
uint16_t writeAddress, uint8_t
writeData) {

Wire.beginTransmission(chipAddress);

  Wire.write(writeAddress >> 8);
//Top 8
  Wire.write(writeAddress & 0xFF);
//Bottom 8
  Wire.write(writeData);
//Write Data
  Wire.endTransmission();
//Bye!
  delay(5);
}
```

The final software challenge is Processing integration. Luckily, Processing is not too difficult to integrate into Arduino programs. All that needs to be done is the Arduino Serial Monitor must be closed and there must be the necessary framework in processing to read serial data, as shown to the below.

```
import processing.serial.*;
Serial port;
String val;     //Data received from the serial port

void setup() {
  port = new Serial(this, "/dev/cu.usbserial-A10L64ER", 9600);
//Note that /dev/cu. is followed by whatever the port name in the specific case
}

void draw() {
  if ( port.available() != 0) {
    val = port.readStringUntil('\n');
//Read until a new line, which in ASCII is '\n'
    if (val != null) {
      //Add code here
    }
  }
}
```

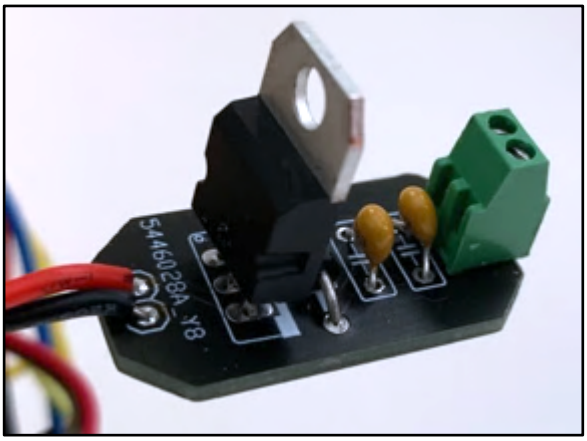
Below is an image of the graph of the temperature data collected from this project

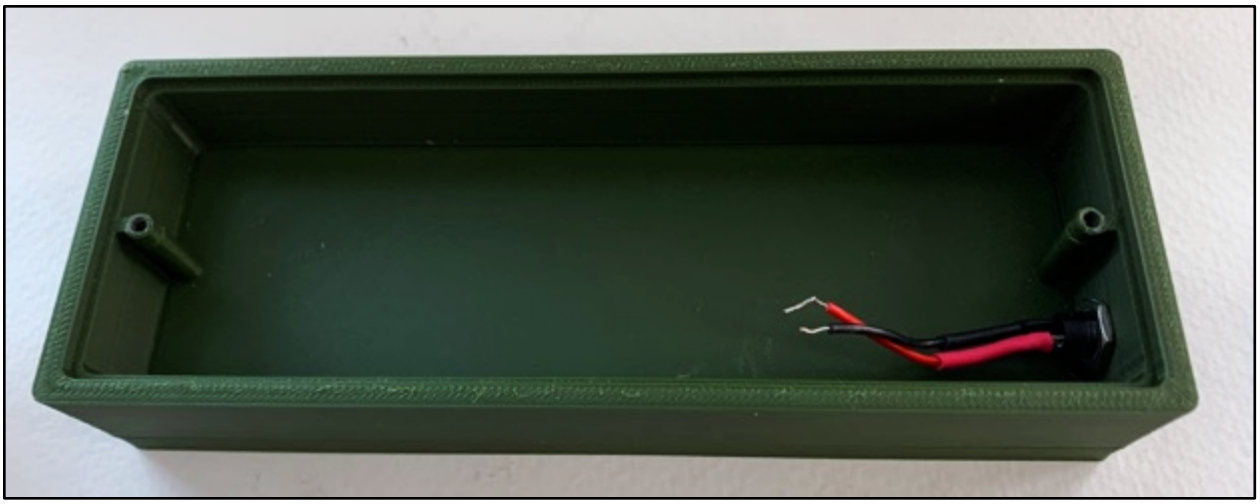



Finally, the last question that needs answering is not purely software, but it is related. That is the question of how often should the Data Logger be storing a sample of temperature. The answer depends completely on the storage size of each data point. Conventionally, each data point stored as hh:mm:ss and temp Celsius would take a minimum of four bytes of memory. However, since the polling rate will be triggered by the DS1307 SQW and we will know the time between each interrupt, the interrupt count itself will give a measurement as to how much time had passed. Assuming that the polling is less frequent than 1.32 seconds (seconds in day ÷ 16-bit integer limit), a data point stored with a two-byte timestamp and the temperature will take only three bytes. Seeing as there is a hard limit on less frequent than every 1.32 seconds (or else the timestamp will be four bytes), the next multiple of the one second interrupt from the DS1307 is two seconds. Therefore, with a data point size of 24 bits and a polling rate of once every two seconds, 129,600 bytes of storage will be needed for 42,000 data points, which will take five EEPROMs assuming they are exactly 256,000 bits each.

## CAD & CAM

This project marks the first instance of two big computer assisted design (CAD) and computer assisted manufacturing (CAM) items. These are custom 3D printed cases designed in Fusion360, and custom PCBs designed in Eagle. This project has a special case printed for the Perma-Proto board and to house the DS1307 board. In addition, the PCB briefly mentioned in the previous project is used to convert a 9 V input to 5 V for the microcontroller. To the right is an image of the PCB, and on the next page is an image of the case.



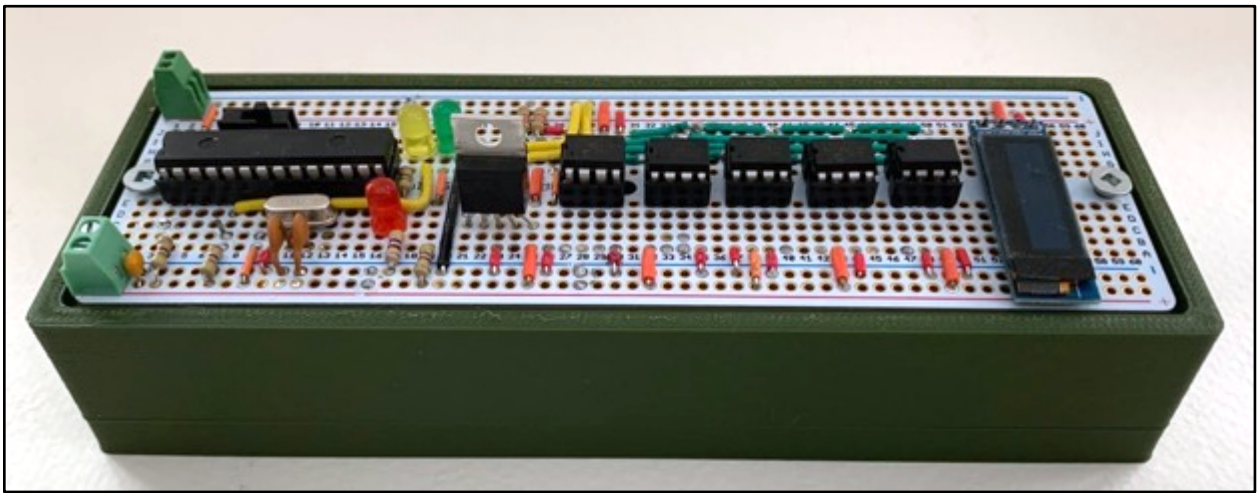


Below is a final image of the Data Logger on a Perma-Proto in a 3D printed case. To the right is the parts table of the final circuit. Note that this does not include the parts to the custom voltage regulator PCB used within the case.

Since this would be a good time to mention it, this circuit uses an ATmega328p. This could be called excessive for the relatively few pins used, but it is actually because of interrupts. The ATtiny84 is not compatible with the AVR core interrupt library. The ATtiny84 does have interrupts, but they are configured at the register level, adding another degree of complexity to an already demanding project. So instead of going to that trouble, and ATmega328p is used.

**Parts Table**

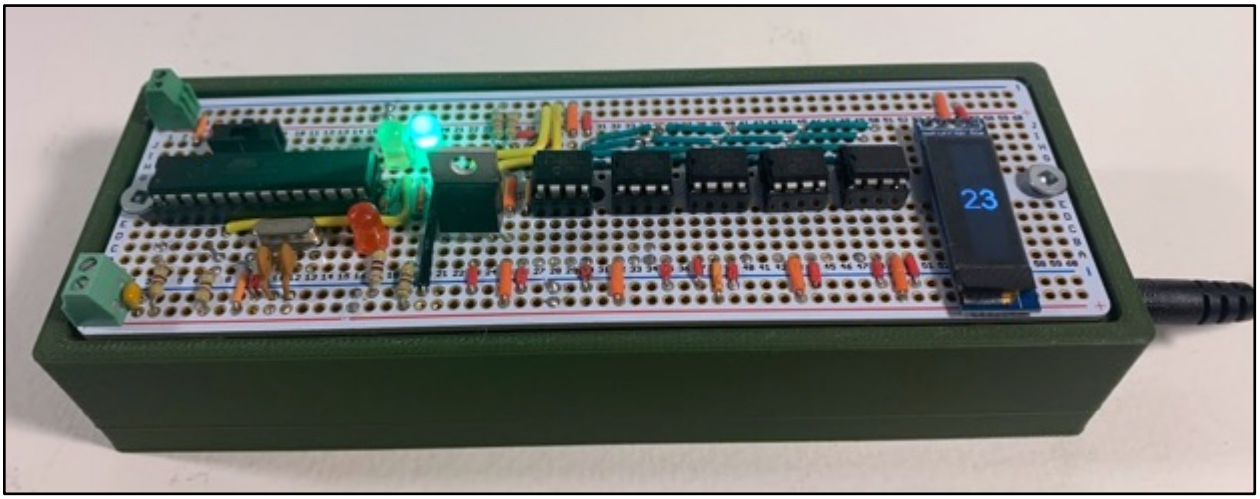
| Description | Quantity |
|---|---|
| ATmega328p | 1 |
| 16 MHz Crystal Oscillator | 1 |
| 20 pF Capacitor | 2 |
| Slide Switch | 1 |
| DS1307 Break-Out-Board | 1 |
| 24 LC256 EEPROM IC | 5 |
| TC74 Temperature Sensor | 1 |
| SSD1306 OLED Display | 1 |
| Adafruit Perma-Proto Board | 1 |
| Assorted LEDs | 3 |
| 3D Printed Case | 1 |
| Custom Voltage Regulator PCB | 1 |
| 2.2 kΩ Resistors | 5 |
| Jumper Wires | ** |



Completed Data Logger Circuit in Case

## Media

Project Video (Extended): https://youtu.be/cQZVER5-5UQ
Project Video (Two Minutes): https://youtu.be/ls5EYLwWhHQ
GitHub: https://github.com/Liam-McCartney/Hardware/tree/main/Data%20Logger




Final Product (OLED Display On)

## Code

Arduino Code:

```
// PROJECT    : I2C Data Logger
// PURPOSE    : Logging and displaying temperature data over a 24h period
// AUTHOR     : Liam McCartney
// DATE       : 26/02/23

// REFERENCE  : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#datalogger
//              https://www.instructables.com/Driving-an-OLED-SSD1306-Display/

// Full Code Available at GitHub:
//https://github.com/Liam-McCartney/Hardware/tree/main/Data%20Logger

#include <Wire.h>

char firstDigitArray[256], secondDigitArray[256];
//Placeholder Arrays for displaying on OLED

unsigned char x, y;
unsigned int z;
//Variables for writing on OLED display

#define RTCADDRESS 0x68
#define completePin 8     //Pin Defintions
#define TC74ADDRESS 0x4D
#define interruptPin 3

uint8_t dump;
//if eepromRead is being used to print and no return is needed then 'dump' is set
the the output

boolean STOP = false;
//Used to stop the programm once 24h has elapsed
uint8_t EEPROM = 0x50;
//Base EEPROM address, will be incremented throughout the programme
```

```
uint32_t interruptCount = 0;
//This is the value that stores the time of the current interrupt
uint16_t address = 0x03;
//First address to hold temp/time data is address 3 as the first 3 are used for
initial time
uint16_t timeStamp = 0;
//This will be used to store the time of the current address being written to

void setup() {
  Serial.begin(9600);

  pinMode(A2, OUTPUT);  //Switch Pins
  pinMode(A0, OUTPUT);

  digitalWrite(A2, 0);  //More switch pins
  digitalWrite(A0, 1);

  boolean state = digitalRead(A1);
  while (digitalRead(A1) == state)
    ;
  /*
  While the switch does not change, the code does not progress. This makes it so
  that EEPROM is not editted when the ATmega328p gets power, essentially allows for
  a Nano to read the EEPROMs without the mega editting them
  */

  attachInterrupt(digitalPinToInterrupt(interruptPin), increment, CHANGE);
  //Setting up interrupt

  pinMode(completePin, OUTPUT);

  Wire.begin();
  config_lcd(); //Runs code that configures the OLED

  //Configuration code from: https://www.instructables.com/Driving-an-OLED-SSD1306-
Display/

  Wire.beginTransmission(RTCADDRESS);
  Wire.write(0x07);       //Edit SQW Config. register
  Wire.write(0b10010000); //Set the SQW to a 1 Hz frequency
  Wire.endTransmission();

  Wire.beginTransmission(RTCADDRESS);
  Wire.write(0);
  Wire.endTransmission();
  Wire.requestFrom(RTCADDRESS, 7); //Requesting all Time info
  while (!Wire.available())
    ;

  uint8_t hours;
  uint8_t minutes;
  uint8_t seconds;
  //Used for initial time-grab

  getTime(hours, minutes, seconds);
  //Sets these variables to the current time

  hours = bcd2dec(hours);
  minutes = bcd2dec(minutes);
  seconds = bcd2dec(seconds);
  //Converts values to decimal (binary acutally)

  eepromWrite(EEPROM, 0x00, bcd2dec(hours));
```

```
  eepromWrite(EEPROM, 0x01, bcd2dec(minutes));
  eepromWrite(EEPROM, 0x02, bcd2dec(seconds));
  //Write the start time to EEPROM
}

uint8_t bcd2dec(uint8_t bcd) {
  return 10 * (bcd >> 4) + (bcd & 0xF);
  //Simple bcd to decimal algorithm
}

void getTime(uint8_t &hr, uint8_t &min, uint8_t &sec) {
  sec = Wire.read();
  min = Wire.read();
  hr = Wire.read();
  //Returns without return statement
}

void loop() {
  if (!STOP && (interruptCount % 4) == 0) {
    //Trigger this if statement every fourth interrupt (every 2 seconds)
    boolean startRead = digitalRead(3);
    //Storing the start read

    if (address == 31998) {
      address = 0;
      EEPROM++;
      //Chip roll-over check
    }

    Wire.requestFrom(TC74ADDRESS, 1);
    while (Wire.available() == 0)
      ;
    int8_t celsius = Wire.read();
    //Get Temp data

    float displayTemp;
    digitalRead(A1) ? displayTemp = celsius : displayTemp = celsius * 1.8 + 32;
    //Depending on the state of the switch, the temp is displayed as C or F
    int8_t displayTempInt = int8_t(displayTemp);
    //convert float to integer

    display(displayTempInt); //Display temp on OLED

    eepromWrite(EEPROM, address, celsius);
    eepromRead(EEPROM, address, dump);
    //Store Temp Data in EEPROM
    address++;

    eepromWrite(EEPROM, address, timeStamp >> 8);
    uint8_t highByte;
    eepromRead(EEPROM, address, highByte);
    //Store highByte of the 2-byte timestamp in EEPROM
    address++;

    eepromWrite(EEPROM, address, timeStamp & 0xFF);
    uint8_t lowByte;
    eepromRead(EEPROM, address, lowByte);
    //Store lowByte of the 2-byte timestamp in EEPROM

    uint16_t collatedStamp = (highBit << 8) | lowBit;
    //Condense the two bytes into one 2-byte number

    if (collatedStamp >= 43200) STOP = ~STOP;
```

```
    //If over 24h has passed, stop is set to true
    address++;
    timeStamp++;

    while (digitalRead(3) == startRead)
      ;
    //Prevents multiple logs in one interrupt

  } else if (STOP) {
    //If the period has passed
    digitalWrite(completePin, 1);
    //Turn on Complete LED
    while (true)
      ;
  }
}

void eepromWrite(uint8_t chipAddress, uint16_t writeAddress, uint8_t writeData) {
  Wire.beginTransmission(chipAddress);

  Wire.write(writeAddress >> 8);    //Top 8
  Wire.write(writeAddress & 0xFF);  //Bottom 8

  Wire.write(writeData);   //Write Data
  Wire.endTransmission();  //Bye!

  delay(5);  //Let the chip finish its write cycle
}

void eepromRead(uint8_t chipAddress, uint16_t readAddress, uint8_t &returnVar) {
  Wire.beginTransmission(chipAddress);

  Wire.write(writeAddress >> 8);    //Top 8
  Wire.write(writeAddress & 0xFF);  //Bottom 8

  Wire.endTransmission();

  Wire.requestFrom(chipAddress, 1); //Send read request
  while (!Wire.available())
    ;
  int8_t dataRead = Wire.read(); //Set datRead to the desired value

  returnVar = dataRead;
  //Return without return function

  Serial.println(dataRead);
}
```

Processing Code:

```
import processing.serial.*;
Serial port;    // Create an object from Serial class
String val;     // Data received from the serial port

final int Length = 1200;
final int Height = 800;    //Height, Length, and Indent
final int indent = 30;

void setup() {
  size(1200, 800);    //Set Window Size
  background(255);    //Set background colour
  stroke(0);          //Set stroke colour
  frameRate(30);
```

```
  port = new Serial(this, "/dev/cu.usbserial-A10L64ER", 9600);
  //Make the port

  line(indent, 0, indent, Height);
  line(Length, Height - indent, 0, Height - indent);
  //Axis

  fill(0);
  textSize(15);

  text("50", indent / 2, indent / 2);

  text("25", indent / 2, Height - Height / 2 - indent / 2);
  text("0", indent / 2, Height - indent / 2);

  text("12h", Length - Length / 2 + indent, Height - indent / 2);
  text("18h", Length - Length / 4 + indent, Height - indent / 2);
  text("6h", Length - 3 * Length / 4 + indent, Height - indent / 2);
  //Axis labels
}

int x = indent;
int intVal;
int n;

void draw() {
  if ( port.available() != 0)
  { // If data is available,
    val = port.readStringUntil('\n');         // read it and store it in val
    if (val != null) {
      val = trim(val);
      intVal = int(val);
      //Convert the read serial data to an int

      float floatVal = map(intVal, 50, 0, Height, 0);
      //Map the int

      point(x++, Height - floatVal + indent);
      //graph the int
    }
  }
}
```

## Reflection

This project definitely took a long time. I would not be surprised if I have spent over 50 hours on it over the last month. Especially without being able to work on it during the weekends except for maybe three hours, it was a bit of a struggle to complete in time. Overall, I am quite happy with the result. While the case is bulky, it was my first ever print, and I am incredibly happy with the PCB inside that came from JLC before this project was due. I also learned a lot from my exploration of Processing and from the OLED display.

On the time management side, this was the first project that ate into my other activities. Fortunately for my other classes and unfortunately for me, sleep was the first thing on the list to be sacrificed. I do not think that I have gone to bed before 12:30 in at least five days, and I will not be changing that tonight. However, I do not necessarily think this is a reflection of my time management, more so a known drawback to taking a weekend job. And really, as long as I am submitting before the deadline, it cannot be so bad.

# Project 2.3c Perma-Proto ATmega328P With PCB

## Purpose

The purpose of this project is to finalize the design from project 2.3a by creating a PCB which is then mounted onto the Perma-Proto board from project 2.3b. Overall, this project encourages exploration into PCB design and incorporating PCBs into pre-existing circuits.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#PermaProtoMegawithPCB
Reference Keypad Algorithm: http://darcy.rsgc.on.ca/ACES/TEI3M/KeypadScanning2223.docx
JLCPCB (For Rendering): https://jlcpcb.com

## Procedure

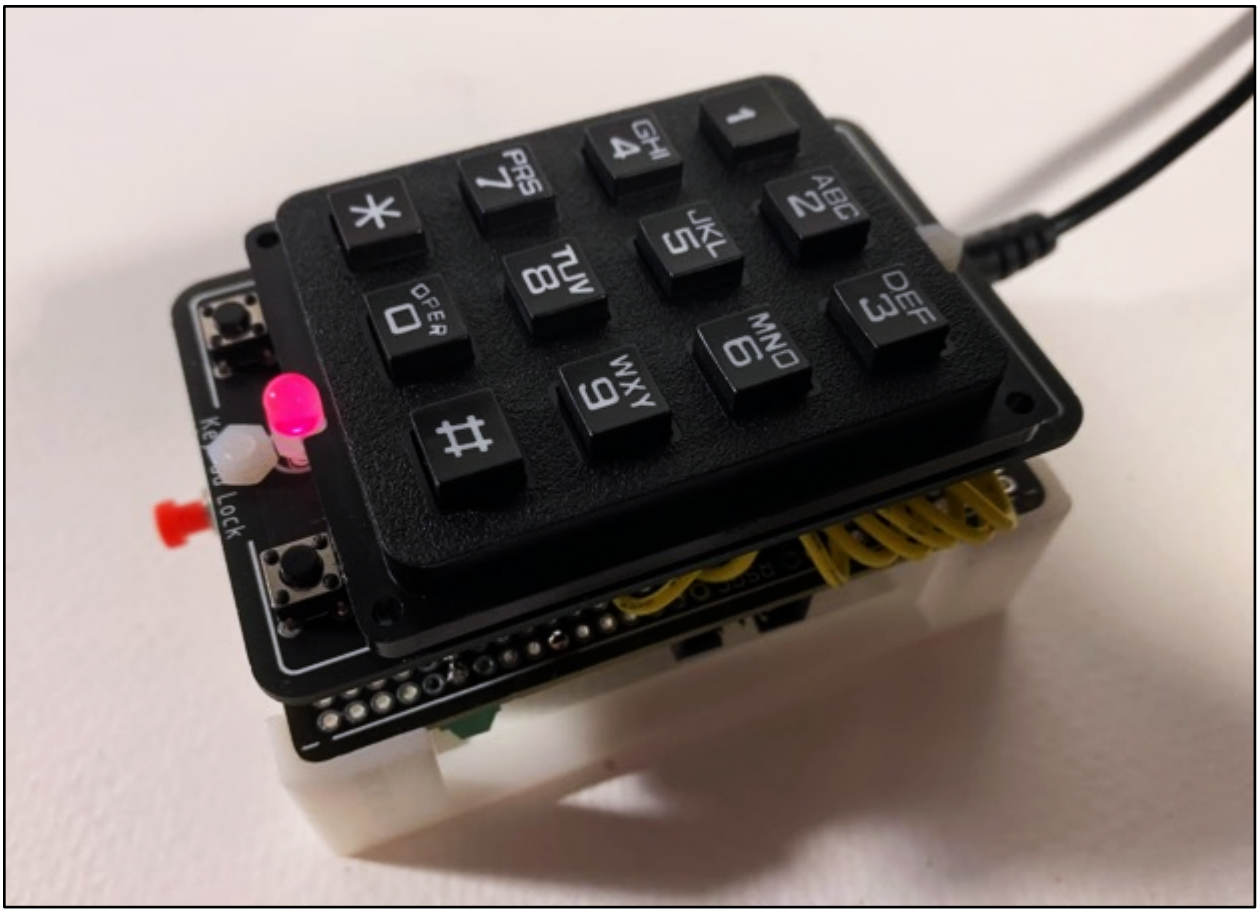
This project can be broken up into two main components: the design, and the software. While the breadboard version of this project already had working code, it has been revised for the PCB version. In addition, the main component, the keypad, is different than in the first version. As for design, this encompasses the PCB creation and mounting onto the Perma-Proto board.

## Design

The PCB design for this project includes three main parts. The keypad, the bicolor LED, and the buttons. So as to make the entire product as sturdy as possible, the PCB is designed to be screwed onto the case along with the Perma-Proto board. The PCB has the exact same dimensions as the Perma-Proto to make this easier. This PCB employs a similar technique as the previous one in the WireFrame Rotator where each component of the board has two sets of holes, one to connect to the ATmega328p on the Perma-Proto and one for the component itself. A picture is shown to the right of the main PCB. It was designed in EAGLE and manufactures by JLCPCB.


The keypad used in the lock covers all of the exposed solder joints of the pins connected to the microcontroller in addition to the two resistors. The only visible parts of the lock, when assembled, are the keypad, the buttons, and the bicolor LED. In addition, there is a reset button mounted in the 3D-printed case.

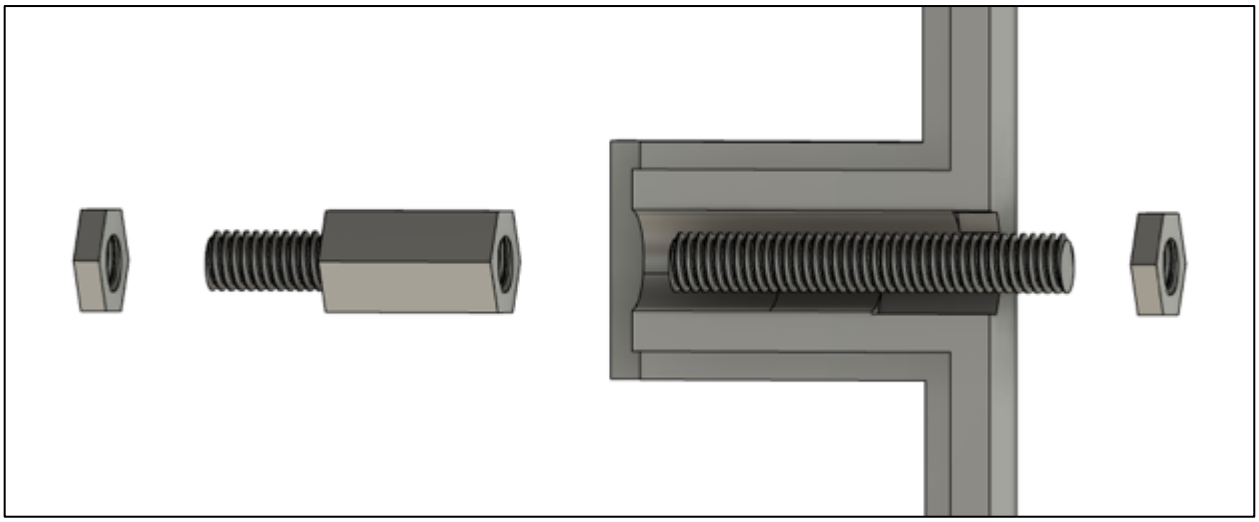
Originally the PCB was meant to use wires directly soldered into the Perma-Proto. However, this would have forever attached the two boards which is quite an undesirable outcome. This decision was due to the sometimes-poor connections when using female headers, but in the end, the final version uses female headers on the Perma-Proto with male headers attached to wires coming from the PCB, as can be seen in the image on the next page.




While it may look like there is quite a bit of spare room at the top (right) of the board, it is important to remember that the screw must be able to freely turn to attach the PCB to the Perma-Proto. There may be a full centimetre of space with no components, but there is only a millimetre or so between the edge of the screw and the start of the keypad. While it is definitely close, the whole contraption does fit in the footprint of a half-sized Perma-Proto board. Much attention was needed to ensure everything fit, going as far as lining up the indent in the keypad with the placement of the bicolor LED. Luckily, the width of the keypad and Perma-Proto are exactly the same, making a nice clean fit.

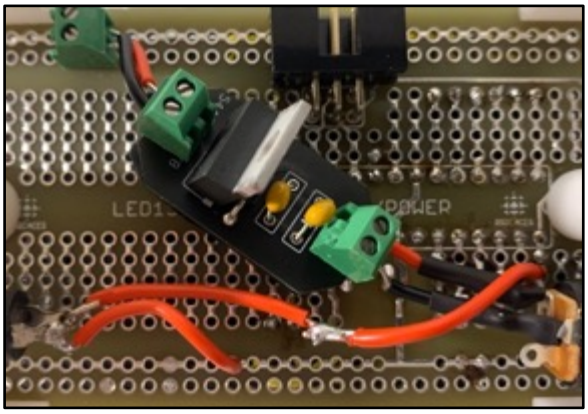
| **Parts Table** | |
|---|---|
| RSGC ACES Perma-Proto 1/2 Sized PCB | 1 |
| ATmega328p Microcontroller | 1 |
| 16 MHz Quartz Crystal Oscillator | 1 |
| 28-Pin Kinked IC Socket | 1 |
| 2 × 3 Shrouded ISP Header (Wurth) | 1 |
| RSGC ACES 1/2 Size Perma Proto Mount | 1 |
| 2.1mm×5.5mm Schurter Power Jack | 1 |
| 12 Key Keypad | 1 |
| Push Button Normally Open | 2 |
| 680 Ω Fixed Resistor | 2 |
| Bicolor LED | 1 |
| Coloured PBNO with Leads | 1 |
| Female Headers | ** |
| Jumper Wires | ** |

The next worthwhile topic to discuss is how the boards are actually attached to the 3D-printed case. While using 30 mm M3 screws and a spacer between boards with a nut in the bottom socket is theoretically possible, it does not work so well in theory. When using a 30 mm screw, the screws go through a lot of abuse while attaching the boards and the heads of the screws are prone to breaking off the shaft. Instead, the method depicted below is used.



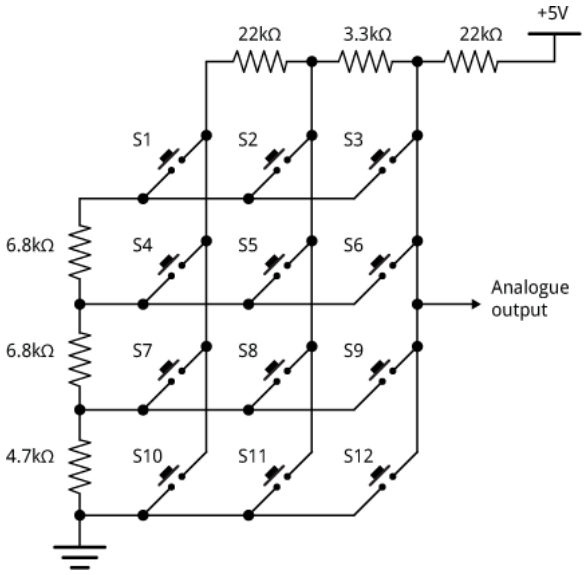
Above is a cross-sectional view of the case for this project in Fusion 360. To attach the two boards, a total of four parts are used per hole. To begin with, a 20 mm screw has its head snipped off so that it is turned into a cylinder with threading, as shown. It is then screwed into the nut on the right in the bottom socket of the case. With the top of the threading poking out the top of the case, the Perma-Proto board is placed in the case and the male-female screw header is then screwed onto the top of the shaft to clamp the board in place. Finally, the PCB rests on the top of the female section of the header, and a final nut secures the PCB. This setup not only takes the stress off the screw but also allows for an extra 5 mm of clearance which avoids the issue of the PCB and Perma-Proto board needing to bend around components to be screwed in place.

In addition to the main PCB in this project is a 5 V regulator PCB, as shown to the right. While this regulator PCB has been used before in the Data Logger, it was originally designed for this project in 2.3b. The regulator makes use of terminal blocks so that it can be permanently secured while still being removable and reusable. As originally predicted, this specific PCB has come in handy multiple times and is much easier to use than wiring up a voltage regulator. That is the true power of PCBs. The regulator is wired on the underside of the Perma-Proto and allows for a 9 V barrel jack to be used as an input, just as the original strip-board version.



## Software

While this is the software section, this is a case where to understand the software you must first understand the hardware of the components being controlled by code. In this case it is the keypad, which although it has been used before there has never been an explanation of how it works. To the right is a schematic. The keypad is essentially a matrix of buttons where each button can be assigned a row and a column, and the seven functioning pins of the device each correspond to either a row or a column. While the schematic is not perfect as is not interfaced with a microcontroller directly, it shows some important aspects. The most important thing is that either the rows or the columns must be either pulled high or low at all times, and it is the group that is pulled in either direction that will be used to read the key being pressed.

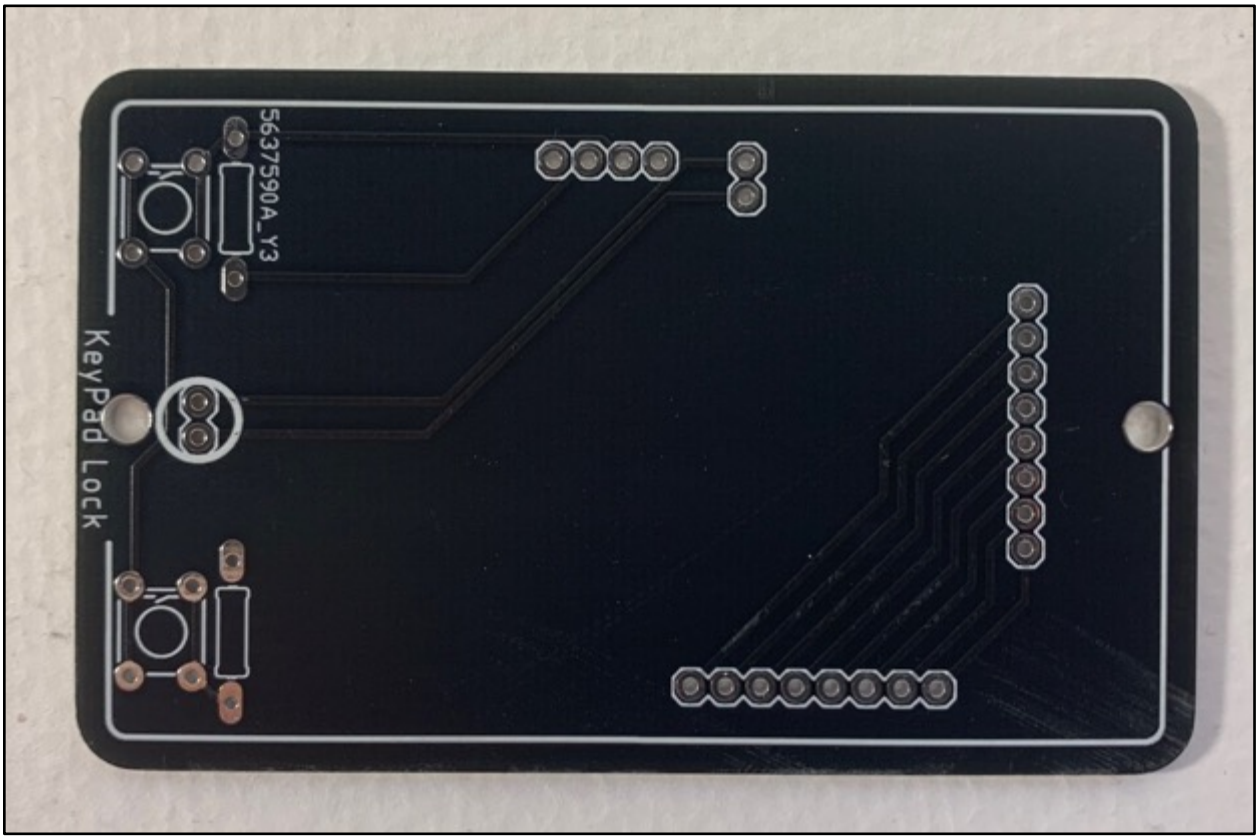



Unlike the breadboard version of the keypad lock, this permanent version does not make use of any external libraries to find the key being pressed. The algorithm is shown with pseudo code to the right. To begin with, some basic information is needed. This includes the number of rows and columns, as well as the pinout of the keypad and which pins are connected to the row and column pins on the device. Next, in setup, the row pins are set to inputs but are pulled up using an internal resistor in the ATmega328p. The columns are set to outputs. To find the key, we first find the row of the key in the matrix of buttons by finding the one row that is being pulled low by a column. That shows us the row of the key. From there, we search for the column. To do that, all columns are set high so that they do not interfere with the button's internal pullups. After that, each column is cycled through, one is set low at a time. From that, we can determine that the moment the row we found earlier reads as low, the column is the only column that has been toggled low. Finally, the function returns the key which is found from an array that is not included in this snippet but is essentially a lookup table for key characters.

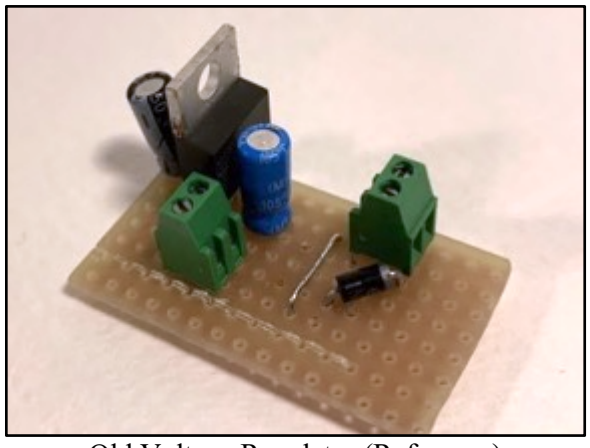
```
#define ROWS 4
#define COLS 3
int rowPins = { 1, 2, 3, 4 };
int colPins = { 5, 6, 7 };

void setup() {
  pinMode(rowPins, INPUT_PULLUP);
  pinMode(colPins, OUTPUT);
}

void loop() {
  char key = getKey();
}

void getKey() {
  for (int i = 0; i < ROWS; i++) {
    if(row = 0) int row = i;
    //Low row is the correct row
  }
  for (int i = 0; i < COLS; i++) {
    digitalWrite(COLS, 1);
    digitalWrite(currentCol, 0);
    //turn all cols high except one
    if(rowPins[row] = 0) int col = i;
    //if row is low, we know the col
  }
  return (keys[row][col]);
  //return key from character array
}
```

## Media

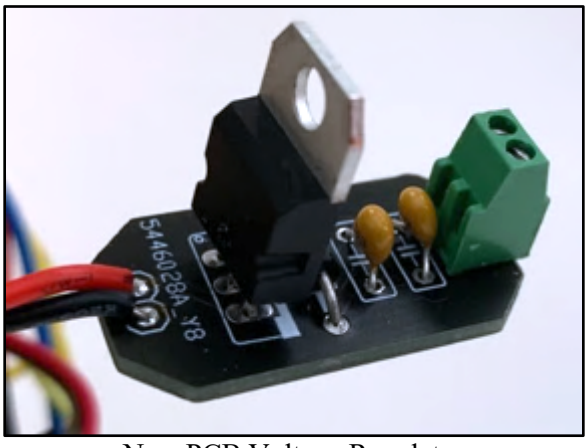
Project Video: https://youtu.be/3FssN-kdy04
Project GitHub: https://github.com/Liam-McCartney/Hardware/tree/main/KeyPad%20Lock%20(2.3c)




KeyPad Lock PCB



Old Voltage Regulator (Reference)




New PCB Voltage Regulator

## Code

```
// Project    : 2.3 Keypad Password
// Purpose    : A device capable of remembering a password and checking if it is
correct
// Course     : ICS3U
// Author     : Liam McCartney
// Date       : 2023 04 01
// MCU        : 328p
// Status     : Working
// Reference :
http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#PermaProtoMegawithPCB

#define COLS 3
#define ROWS 4

uint8_t rowPins[ROWS] = { A1, A2, A3, A4 };  //connect to the row pinouts of the
keypad
uint8_t colPins[COLS] = { 9, 10, A1 };       //connect to the column pinouts of the
keypad

char keys[ROWS][COLS] = {
  { '1', '2', '3' },
  { '4', '5', '6' },
  { '7', '8', '9' },
  { '*', '0', '#' }
};

volatile boolean unlocked = false;
char key;

#define length 6
uint8_t password[length] = { 8, 7, 8, 9, 9, 9 };
uint8_t history[length];
//The password is initially preset until the user choses one

#define ledRed 7
#define ledGreen 8  //Pin definitions
#define savePassword 2
#define resetPin 3

void setup() {
  for (uint8_t i = 0; i < ROWS; i++) {
    pinMode(rowPins[i], INPUT_PULLUP);
  }
//Rows are inputs with internal pullups

  for (uint8_t i = 0; i < COLS; i++) {
    pinMode(colPins[i], OUTPUT);
    digitalWrite(colPins[i], 0);
  }
//Cols are outputs

  pinMode(ledRed, OUTPUT);
  pinMode(ledGreen, OUTPUT);
  digitalWrite(ledRed, 1);
  digitalWrite(ledGreen, 0);

  pinMode(0, OUTPUT);
  pinMode(1, OUTPUT);

  digitalWrite(0, 1);
  digitalWrite(1, 1);
```

```
  //Making the bicolor LED red to show the device is "locked"

  attachInterrupt(digitalPinToInterrupt(resetPin), reset, RISING);
  //This interrupt listens the reset pin and calls the reset function
}

void loop() {
  if (unlocked) {
    digitalWrite(ledGreen, 1);
    digitalWrite(ledRed, 0);
    //Set the LED to green to show that the device is unlocked

    key = 0;
    while (key < 48 || key > 57) {
      for (uint8_t n = 0; n < COLS; n++) digitalWrite(colPins[n], 0);
      char key = getKey();
    }
    //Reads the current key pressed (even if it no key is pressed)

    for (int i = 0; i < length; i++) {
      history[i] = history[i + 1];
    }  //Shift ever index in history back 1

    history[length - 1] = key - 48;
    //Add the current key as the last index

    if (digitalRead(savePassword)) {
      //If the save password button IS pressed (on by default, PBNC)
      for (int i = 0; i < length; i++) password[i] = history[i];
      //Make the password the current history
    }
  } else {
    //If not unlocked
    digitalWrite(ledGreen, 0);
    digitalWrite(ledRed, 1);
    //Make the LED red

    key = 0;
     while (key < 48 || key > 57) {
      for (uint8_t n = 0; n < COLS; n++) digitalWrite(colPins[n], 0);
      char key = getKey();
    }
    //Get the current key

    for (int i = 0; i < length; i++) {
      history[i] = history[i + 1];
    }
    history[length - 1] = key - 48;
    //Again shifting history by 1 index then adding current read

    unlocked = true;  //No matter what, the lock unlocks here

    for (int i = 0; i < length; i++) {
      if (history[i] != password[i]) unlocked = false;
    }
    //But it only stays unlocked if it "survives" this for loop
  }
}

void reset() {
  unlocked = false;
}
```

```
uint8_t getRow() {
  for (uint8_t i = 0; i < ROWS; i++) if (!digitalRead(rowPins[i])) return i;
  //If a row is pressed, return the row
  return ROWS;
  //If not, return ROWS which will be out of bounds
}

uint8_t getCol(uint8_t row) {
  for (uint8_t i = 0; i < COLS; i++) {
    for (uint8_t n = 0; n < COLS; n++) digitalWrite(colPins[n], 1);
    digitalWrite(colPins[i], 0);
    //Isolate a single column

    if (!digitalRead(rowPins[row])) return i;
    //Return the column if the row goes low
  }
  return COLS;
  //Otherwise return out of bounds
}

char getKey() {
  uint8_t row = getRow();
  uint8_t col = getCol(row);
  return (keys[row][col]);
  //Return key from lookup array, will be null if no key
}
```

## Reflection

This project was actually quite enjoyable. It was fun to have a relatively simple task at hand (since I had already used EAGLE multiple times before) and be able to just see it through. The one exception to this was the custom code to read the keypad, which I pushed myself to build from scratch. I used the same algorithm that we discussed in class, but aside from that I did it all myself with no external help or research, except for the information on how to set up an internal pull-up resistor. I did this as a challenge and it was quite rewarding after I saw the first correct values coming through the serial monitor, though it took longer than I would like to admit. As for the actual PCB design, I mentioned how I was originally planning to not use female headers, specifically as they did not work with previous projects. I was eventually persuaded that they were reliable enough when using the thicker male headers compared to wire, but this late decision brought me to my one pet peeve of the project where I could have fully eliminated wires if I had committed to headers earlier in the design process. Either way, I am happy with my result of a functioning, clean-looking version of my original concept from January.

# Project 2.20c Medium ISP: The WireFrame Rotator

## Theory

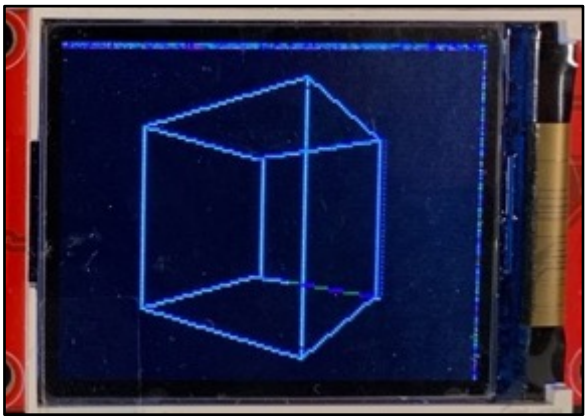
The WireFrame Rotator (WFR), as its name suggests, is a device that rotates wireframes of three-dimensional shapes on a two-dimensional screen. While at first this task may seem somewhat elementary and mundane, the acts of both rotating a point in three dimensions and then translating that point onto a two-dimensional plane proves to be quite intricate. The WFR also stresses customization with a user-interface (UI) which allows for unique wireframes to be displayed on the screen without editing the code. To do this, the setup algorithm must be very adaptable, a very useful attribute in the real world.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/ISPs.html#logs
Deriving the Rotation Matrix:
https://www.youtube.com/watch?v=NNWeu3dNFWA&t=349s&ab_channel=FlammableMaths
Projection Mapping: https://en.wikipedia.org/wiki/3D_projection

## Procedure

As discussed in the theory section, much of this project's complexity comes from the math and its theory. However, one thing is essential to understand everything that is to come: the wireframe. While this term has many meanings, in this project it refers to a 3D object represented by a collection of 2D lines which connect vertices with 3D coordinates. To the right is an image of a wireframe generated by the WFR. This specific wireframe has eight vertices and 12 lines. Note that the image can be customised to be a collection of points on a single 2D plane (essentially a 2D shape when viewed perpendicular to the plane) rotating in three-dimensions as well.



## Math & Theory

The math in this project can essentially be simplified to two broad subjects: rotation and projection mapping. Rotation is self-explanatory, but projection mapping might need an explanation. Projection mapping is essentially the act of representing an object from a higher dimension in some lower dimension. This project uses 3D to 2D projection mapping, where 3D shapes are represented on a 2D screen.

### Rotation

There are multiple ways to approach the problem of rotating a 3D object in 3D space, but the WFR makes use of a loophole that simplifies things. Technically, 3D rotation does not exist. Depending on where you define your axis in 3D space relative to your point being rotated, it is always possible for one value in the $(x, y, z)$ coordinate to stay constant. It is difficult to explain this intuitively, but take for example the cube shown above. While it rotates, the height of each point is conserved, only the $x$ and $z$ coordinates change. If you wanted an off-axis rotation, simply re-orient the vertices relative to the axis and the same rotation will appear off-axis.

This simplification by definition sacrifices the ability to have all coordinate vectors change in a rotation, but since any rotation can still be represented all the same, this hindrance is not significant. To summarize a somewhat confusing idea, the WFR only used 2D rotations where it can be inferred that one component of each vertex will stay constant with the rotation. While there are still ways to deal with un-simplified 3D rotation, this method allows us to use the following rotation matrix:

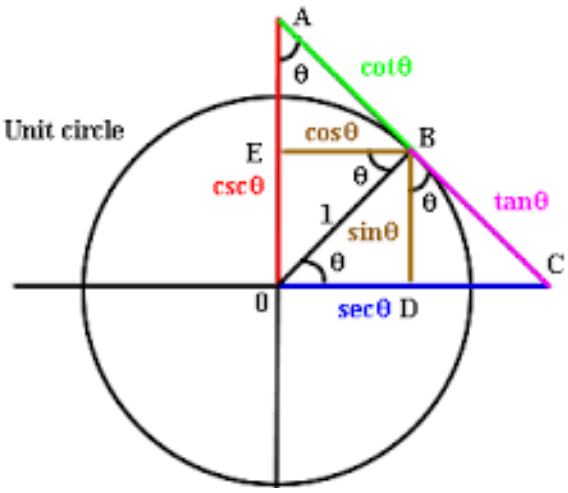
$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & +cos\theta \end{bmatrix}$$

Without yet going into the derivation, a rotation matrix is essentially a way to scale the coordinates of a point so that they lie $\theta$ around a circle centered on the origin. To "use" a rotation matrix, you simply cross-multiply the original coordinates by the rotation matrix to get the new rotated coordinates like so:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & +cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

### Rotation Matrix Derivation

So as to eliminate the potential risk of portraying rotation matrices as a piece of magic, here is an explanation of how they work. For the explanation to make sense, it is helpful to have a trig-ratio circle to look at to visualize each step of the process, as to the right.




The first step of the derivation is to write down the initial state of the point that we wish to rotate. This can of course be done with a simple $(x, y)$ coordinate point, but to make the following steps easier this can be expanded. Firstly, we will not use a point, we will use a vector, as such:

$$\vec{r} = \begin{pmatrix} x \\ y \end{pmatrix}$$

Now we will re-write the vector in terms of trig-ratios. Looking at the trig-circle, we see that $x = cos\theta$ and $y = sin\theta$. In this case, $\varphi$ will be used instead of $\theta$, where $\varphi$ is the angle swept out by the initial state of the coordinate's vector. Assuming that this is to be expanded beyond the unit circle, both values are scaled by the radius of the circle, $r$.

$$\vec{r} = \begin{pmatrix} r \bullet cos\varphi \\ r \bullet sin\varphi \end{pmatrix}$$

This same notation will be used for the coordinates after rotation. Imagine that after the initial position of $\varphi$, another angle $\theta$ was further swept out. The total angle is now equivalent $\varphi + \theta$. This information can be used to write out the rotated vector, denoted with $\vec{r}'$.

$$\vec{r}' = \begin{pmatrix} r \bullet cos(\varphi + \theta) \\ r \bullet sin(\varphi + \theta) \end{pmatrix}$$

This can now be expanded with the trigonometric addition theorems, which essentially state that $sin(x+y) = sin(x)\,cos(y) + cos(x)\,sin(y)$, and that $cos(x+y) = cos(x)\,cos(y) - sin\,(x)sin(y)$. It is also important not to forget that $r$ has been factored out. By expanding, we get the following:

$$\vec{r}' = \begin{pmatrix} r \bullet cos(\varphi)\,cos(\theta) - r \bullet sin(\varphi)sin(\theta) \\ r \bullet sin(\theta)\,cos(\varphi) + r \bullet sin(\varphi)\,cos(\theta) \end{pmatrix}$$

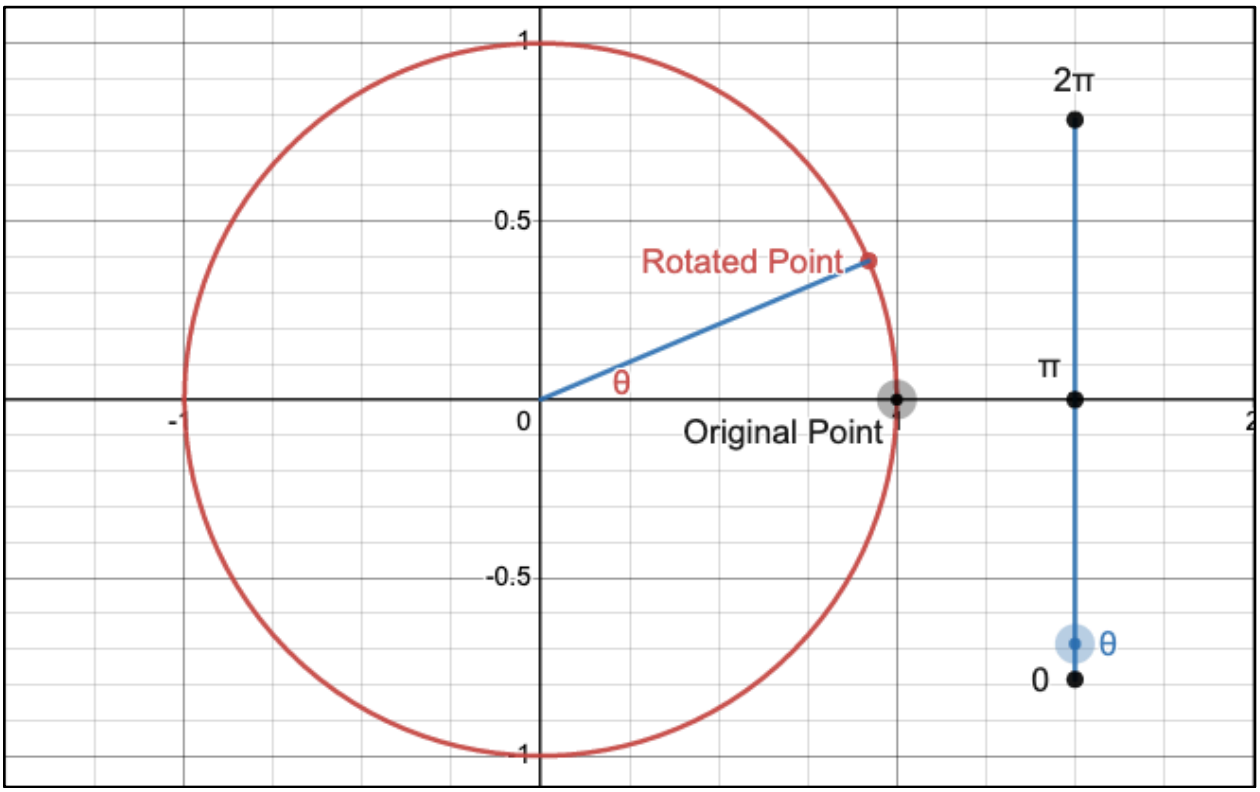
Now, by referring to the second formula of the derivation, we can simplify $r \bullet cos\varphi$ to $x$ and $r \bullet sin\varphi$ to $y$. Rewriting, we get the following:

$$\vec{r}' = \begin{pmatrix} x \bullet cos\,\theta - y \bullet sin\theta \\ x \bullet sin\theta + y \bullet cos\theta \end{pmatrix}$$

With this step, the original formula becomes clear. By looking at the $x$ and $y$ positions, we can rewrite this equation as the original rotation matrix:

$$\vec{r}' = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & +cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

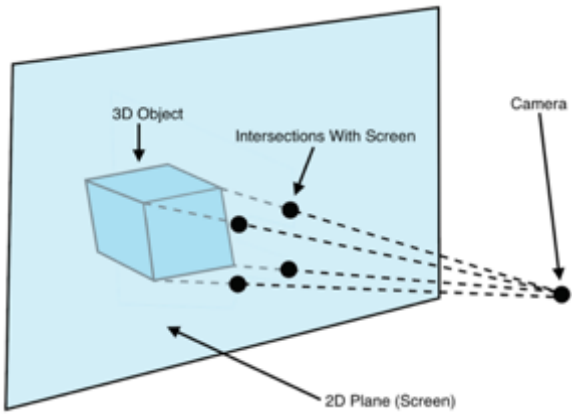
This can be translated to simpler math where $y' = x \bullet cos\theta - y \bullet sin\theta$ and $x' = x \bullet sin\theta + y \bullet cos\theta$, and with this it can be used in a Desmos graph to visualize. Below is a screenshot of the rotation matrix being used in Desmos to rotate a point around a circle.



## Projection Mapping

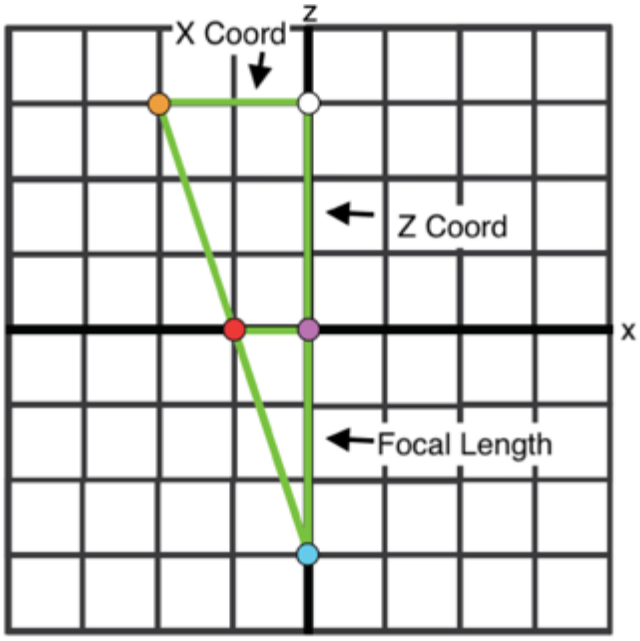
Projection mapping, as previously described, is the act of translating 3D points to a 2D plane. This is required for this project as the wireframe being rotated is 3D while the screen it is being displayed on is 2D. There are many ways to accomplish projection mapping, and they all have their own unique attributes. For example, the WFR uses a method of perspective mapping that somewhat distorts the image. In fact, lines that should be parallel in Euclidian space will meet at some finite point on the WFR. However, this is not necessarily a problem, it actually gives more perspective to the 3D shape on the 2D screen and is widely used for projection mapping.

The act of perspective mapping is essentially finding intersections between dimensions. The diagram to the right shows four vertices of a cube being mapped to a screen. To map an object, you simply imagine lines between a camera at some customizable, fixed position and the points that you wish to map. The $(x, y)$ coordinates of the point of intersection of the lines and 2D plane correspond to the mapped coordinates of the 3D object, or the pixels that should be coloured on the screen. Interestingly, this is the same method used to map 4D objects to 3D space in the many online videos of rotating tesseracts: first the intersections of the 4D lines and the 3D plane are found, then the 3D shadow of the 4D object is mapped to 2D.

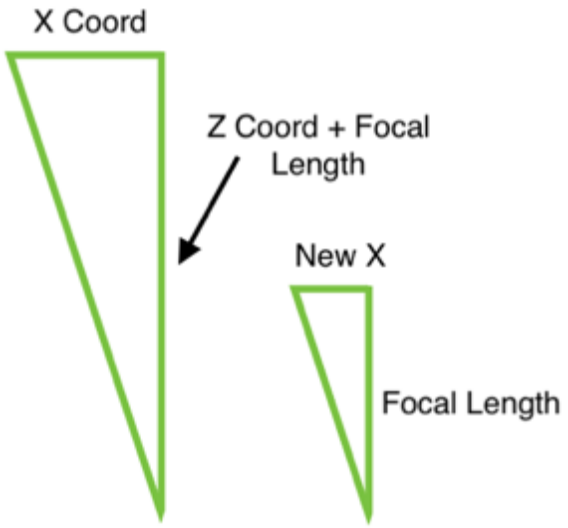



While this works in theory to map points from 3D onto a 2D screen, it is pretty much unusable without a formula that can be used to take 3D points and output 2D screen coordinates. This can be broken down further: the formula must be able to output both an $x$ and $y$ value, but not necessarily simultaneously.

The image to the right shows a bird's eye view of the original example, where the blue point is the camera, the orange point is the vertex being mapped, and the red point is the intersection between the line and the screen. The $x$ axis can be thought of as the screen itself. The side labelled "focal length" is the length between the origin and the camera, and this length is up to the person who is wishing to map the point. The purple and white dots are simply there to construct triangles with. In this diagram there are two right-angle triangles: one bound by the camera, vertex, and the white dot, and the other bound by the point of intersection, the camera, and the purple dot. What is important here is that these two triangles are similar, which means that one can be used to solve for the side lengths of the other.



To the right is a diagram of the two earlier-mentioned triangles. Out of the four labelled values, we know three. The sides labelled $x$ and $z$ coordinates simply have a length of the corresponding coordinates in 3D space. The side that we wish to solve for is *new* $x$, or $x'$. Using basic geometry, we can create a formula to solve for this side using the fact that the ratio of x : $focal\ length\ + z$ is equal to x' : $focal\ length$.



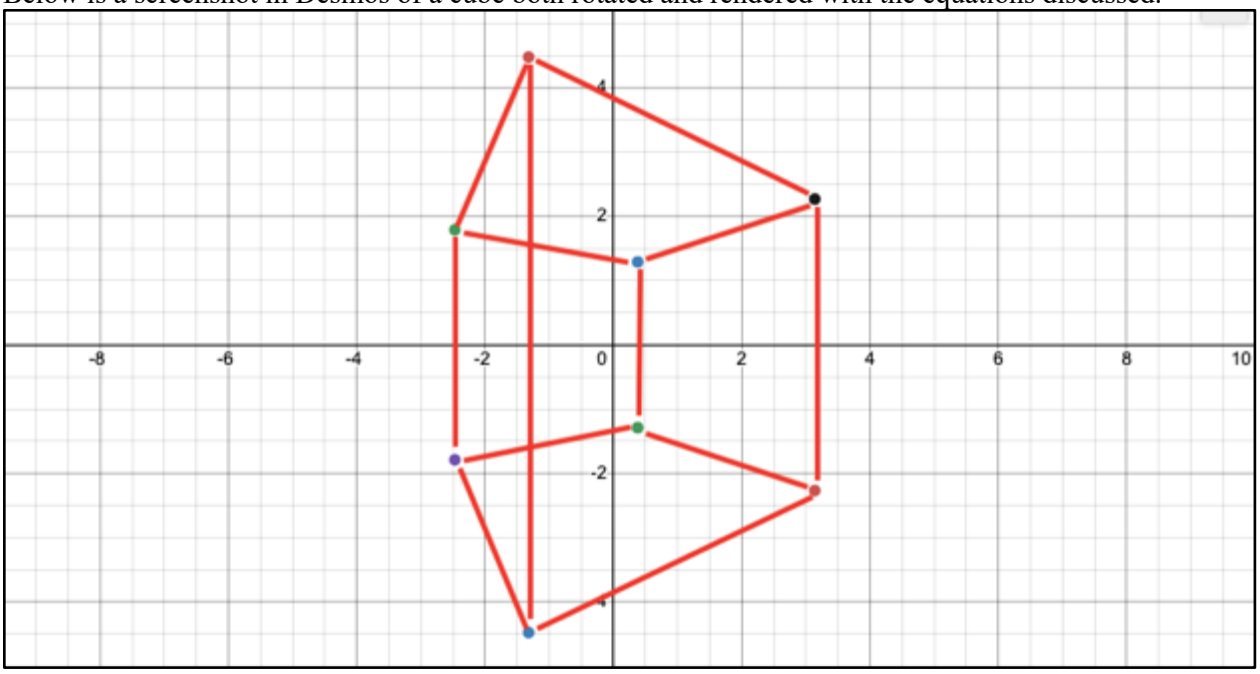

$$\frac{x'}{focal\ length} = \frac{x}{focal\ length + z}$$

Multiply both sides by $focal\ length$ to solve for $x'$:

$$x' = \frac{focal\ length \bullet x}{focal\ length + z}$$

This formula solves for $x'$, but what works out nicely is that the math in a cross-sectional view would work out the exact same, except all instances of $x$ would be replaced with $y$. So essentially to map a vertex of a wireframe to a 2D screen, this formula must be solved twice, once for $x$ and once for $y$. Luckily, edges can be drawn between points that have already been mapped onto the screen, so there is no need to attempt to map a full line to two dimensions.

Below is a screenshot in Desmos of a cube both rotated and rendered with the equations discussed.



## Software

While software has not yet been discussed, it has at the same time been mostly covered. The whole post-setup program of the WFR is essentially just the application of the previous math theory. That being said, the setup itself is still worth mentioning.

With the adaptability of the code, there are still limitations. Due to global variable limitations, the maximum number of edges and vertices is 20. This limitation is mostly due to the fact that for the vast majority of the time, the coordinates for the vertices of the wireframe are non-integer values, meaning they must be stored in floats, using up four bytes per coordinate, plus a set of backup-coordinates, plus another two sets of pre-set coordinates. Essentially, the coordinate points need to have high resolution, and without the implementation of floating-point arithmetic, this takes up a lot of storage.

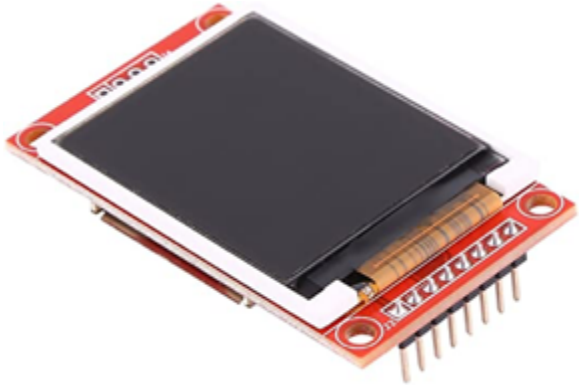
Either way, the program works as follows (example to right): The user inputs the number of vertices that they would like their shape to have. This number is then stored as a variable, n for example. Then, a loop iterates n times and each time requests for a point in 3D space, and each coordinate is stored in a two-dimensional array where each primary index has three secondary indices. Each coordinate then has a value (its index in the coordinate array), this is important later. The user then inputs the number of edges for their shape, and this number is then stored similarly to the number of vertices (as m for example). Then, for m iterations, the used inputs a "edge table", which essentially links indices in the coordinate array with their previously described values in another two-dimensional array.

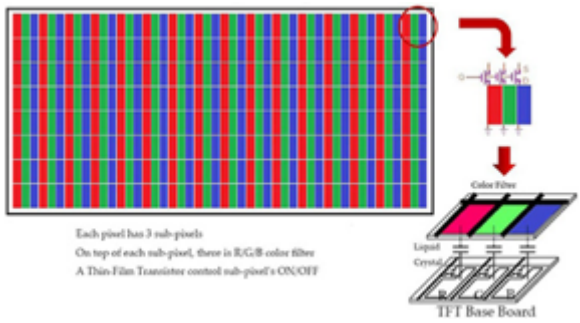
To the right is the full inputted information of a cube, which has eight vertices and 12 edges. One of the difficulties with this method of input is that it is incredibly difficult to visualize what is being inputted to the program. However, there is no real intuitive way to input a 3D shape using only a keypad, and for that reason, pre-set shapes can be loaded onto the microcontroller through the code.

| **Input Number of Vertices (8)** | |
|---|---|
| Index Number | Coordinates |
| 0 | (2, 2, 2) |
| 1 | (2, 2, -2) |
| 2 | (2, -2, 2) |
| 3 | (2, -2, -2) |
| 4 | (-2, 2, 2) |
| 5 | (-2, -2, 2) |
| 6 | (-2, 2, -2) |
| 7 | (-2, -2, -2) |
| **Input Number of Edges (12)** | |
| Edge Number | Indices Connected |
| 0 | 0, 1 |
| 1 | 0, 2 |
| 2 | 0, 4 |
| 3 | 1, 3 |
| 4 | 1, 5 |
| 5 | 2, 3 |
| 6 | 2, 6 |
| 7 | 3, 7 |
| 8 | 4, 5 |
| 9 | 4, 6 |
| 10 | 5, 7 |
| 11 | 6, 7 |

## Hardware

The only new piece of hardware in this project in the ST7735 TFT LCD used to display the final wireframe, but this is no simple piece. It has lots of unexplored usage, such as an SD card reader on the back. To the right is a picture of the display. While the display has eight pins, only seven are needed to function, and out of those seven only three are used for communication. The pin that is not needed is interestingly the 5 V pin. Through testing it seems as though the display runs perfectly fine off of the 3.3 V for the backlight. The display is 128 × 160 pixels.



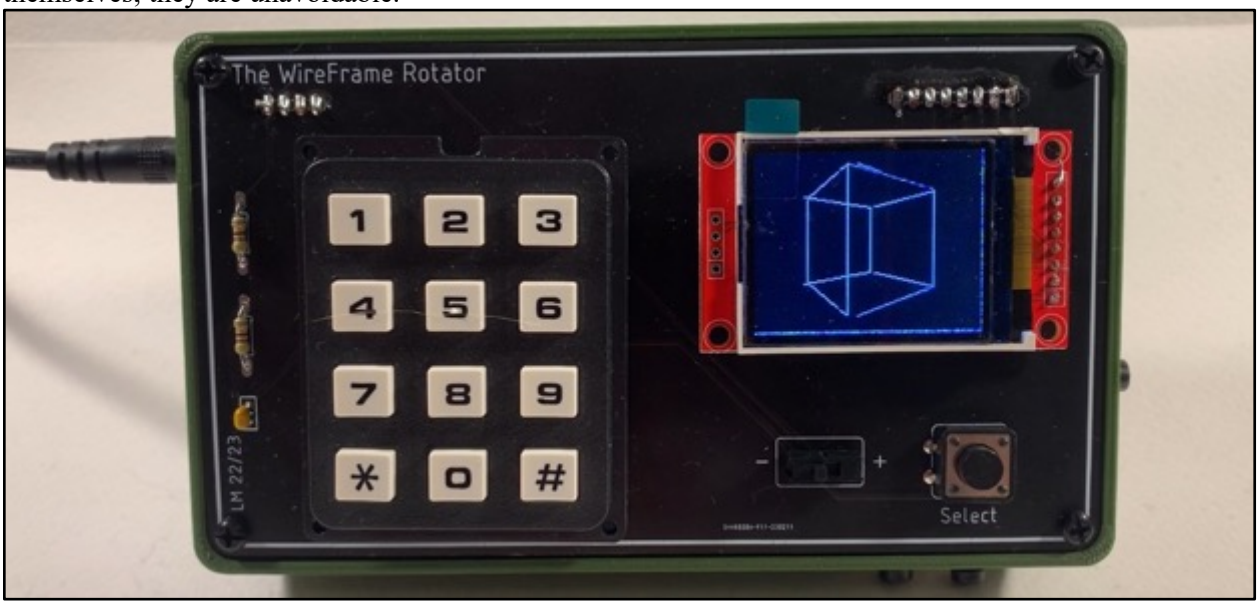



LCDs, or liquid crystal displays, work by either blocking or allowing light to pass through. Simple black-and-white LCDs rely on outside light to reflect off of them to produce a display, but graphic TFT LCDs such as the ST7735 are backlight. Essentially, each pixel is made up of three smaller pixels that can only be red, green, or blue. Depending on the amount of light allowed through each sub-pixel, the overall colour of the pixel is affected. To change the amount of light allowed through the display, voltage can be applied by transistors on each of the RGB sub-pixels to change the chirality of the liquid crystal in the display. Chirality is essentially a chemistry-term for how much something spins light. The top glass of the display has a chirality as well, and when voltage is applied, the glass and liquid crystal together have a chirality that stops a certain amount of light, depending on the voltage.

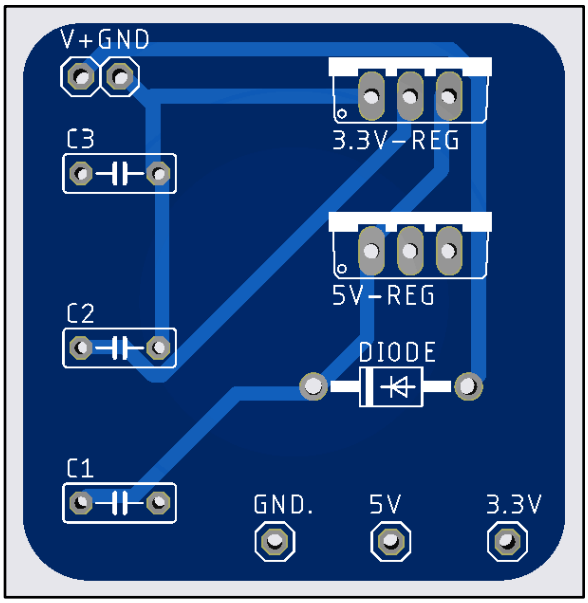
## CAD/CAM & Final Product

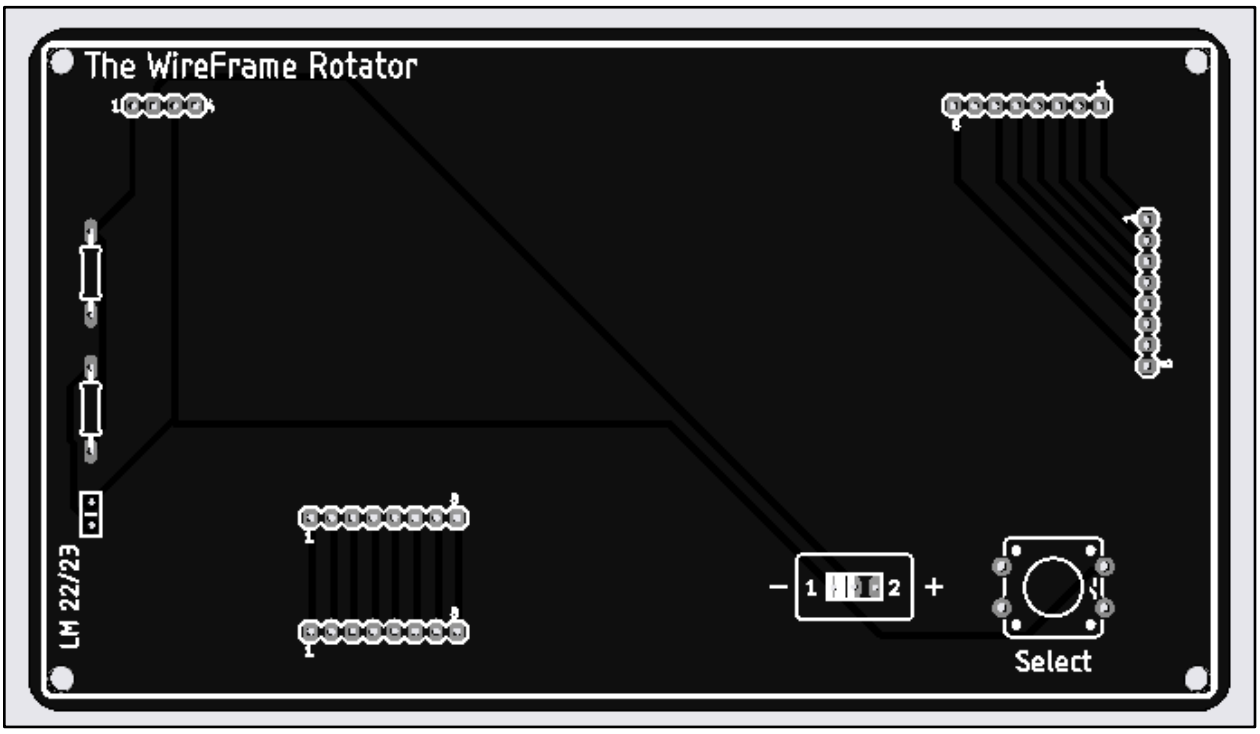
Below is a picture of the final build of the WFR. To the right is the parts list. On the bottom side of the case are the two pre-set buttons, and on the side is the select button. Due to an error in the PCB, the select button on the board itself does not work (the reset button was repurposed for the select button after the error was discovered), but otherwise the device is fully functional. A power-jack is located opposite to the select button. There is no microcontroller on the topside of the WFR, but underneath the PCB is a half-sized Perma-Proto board with an ATmega328p microcontroller on it, along with all the wiring and another PCB for 3.3 V and 5 V regulation. The half-sized Perma-Proto is mounted with screws inside the case. The ribbons of static colour on the sides of the LCD seem to be due to an issue with the displays themselves; they are unavoidable.

| **Parts Table** | |
|---|---|
| Description | Quantity |
| ATmega328p | 1 |
| 16 MHz Crystal Oscillator | 1 |
| 20 pF Capacitor | 2 |
| Slide Switch | 1 |
| Large Button (PBNO) | 1 |
| Screw-On Button (PBNC) | 3 |
| ST7735 TFT LCD Display | 1 |
| 12 Key Keypad | 1 |
| 0.1 µF Capacitor | 1 |
| Half-Sized Perma-Proto Board | 1 |
| 3D Printed Case | 1 |
| Custom Voltage Regulator PCB | 1 |
| Custom WireFrame Rotator PCB | 1 |



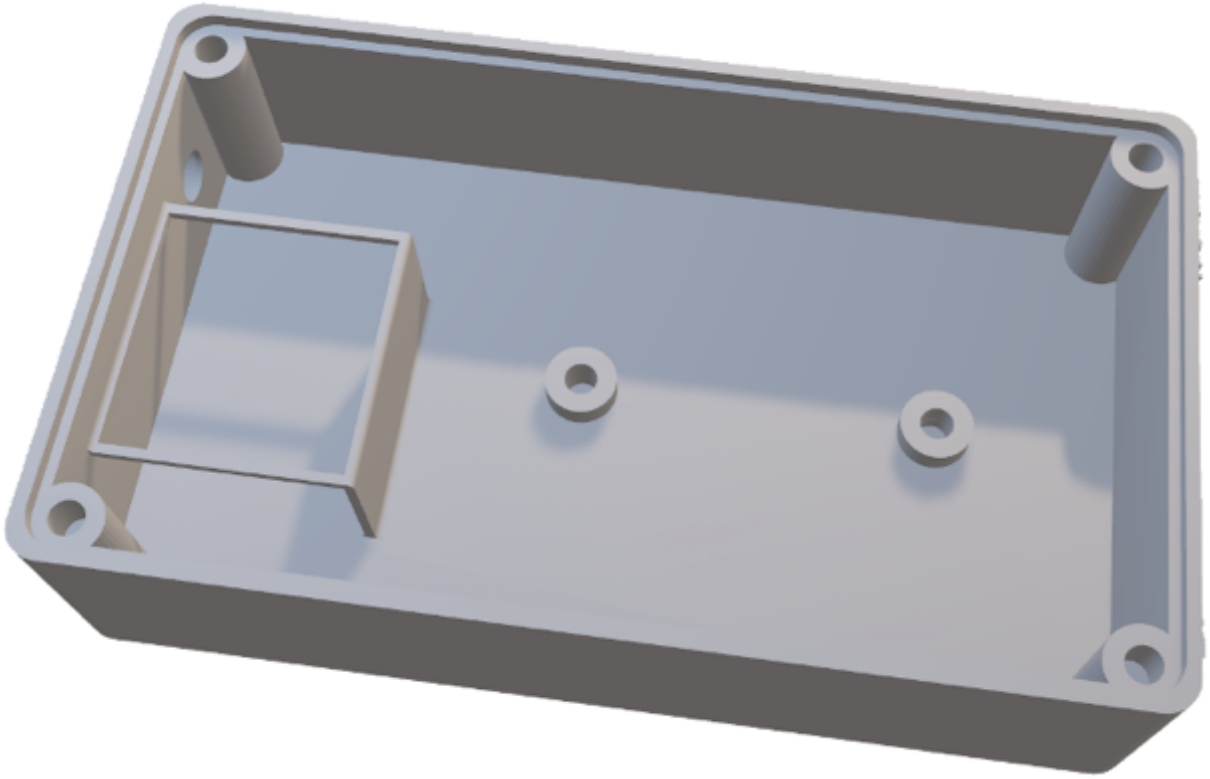
To the right is a picture of the 3.3 V and 5 V regulator. It was designed in Eagle and manufactured by JLCPCB to put both the 5 V and 3.3 V regulator on the same board. The PCB from the Data Logger could not be used in this case since the L7805 (5 V regulator) and the LD33V (3.3 V regulator) have different pinouts. This regulator fits into the roll-cage that will be seen in the 3D printed case design. The barrel-jack's power and ground are attached to V+ and GND through a terminal block so that the PCB may be removed from the case if needed, and the outputs are also all attached to their destination through terminal blocks. The PCB is designed to accept both polarized and non-polarized capacitors. In this case, non-polarized ceramic capacitors were used. Below is a screenshot of the main PCB. It is mostly a break-out-board for the keypad and LCD which need to be oriented with perpendicular pin alignment.



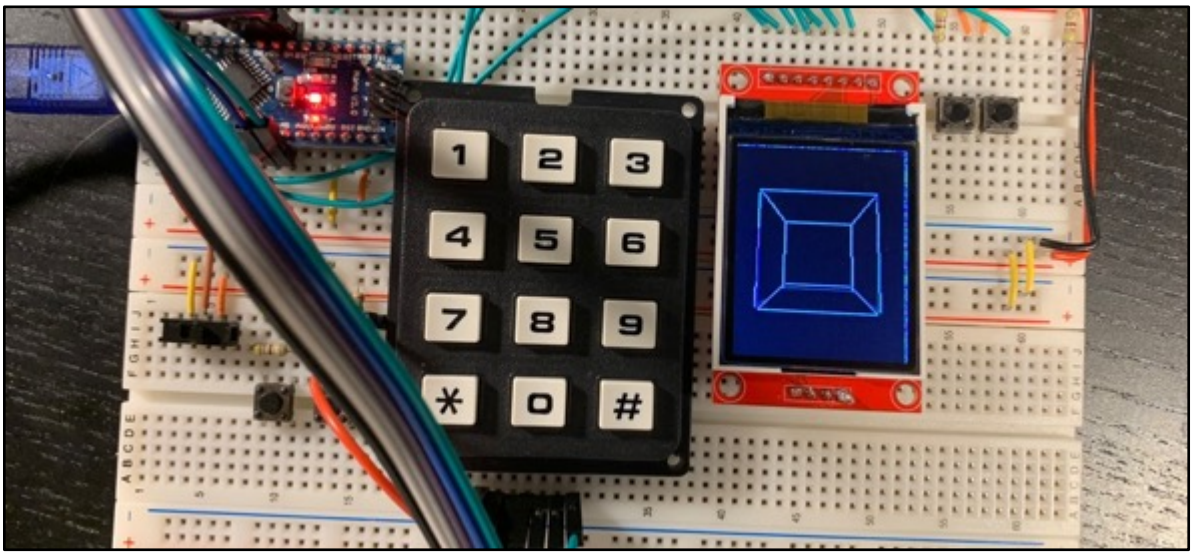

The two main components (LCD and keypad) both have two sets of holes. One set is for ribbons of wire that attach to the microcontroller board, and the other set is for the device itself. As mentioned, the Select button is incorrectly wired and does not work properly due to a short, so while it is soldered on the board, it is floating and the function of the Select button has been repurposed to what was supposed to be the RESET button on the right side of the case.

Below is a screenshot of the 3D printed case's STL file. In the picture, the two holes for mounting the half-sized Perma-Proto are clearly visible, and well as the "roll-cage". The main body of the case is 1 mm smaller than the PCB, while the rim where the PCB sits is 1 mm wider (0.5 mm on each side).

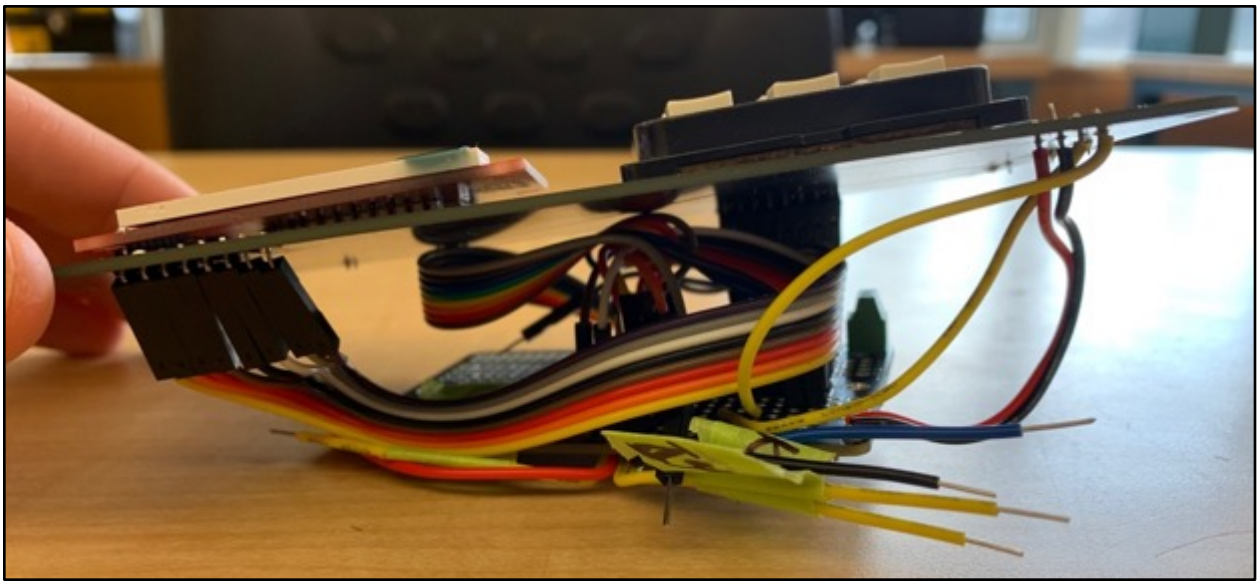


## Media

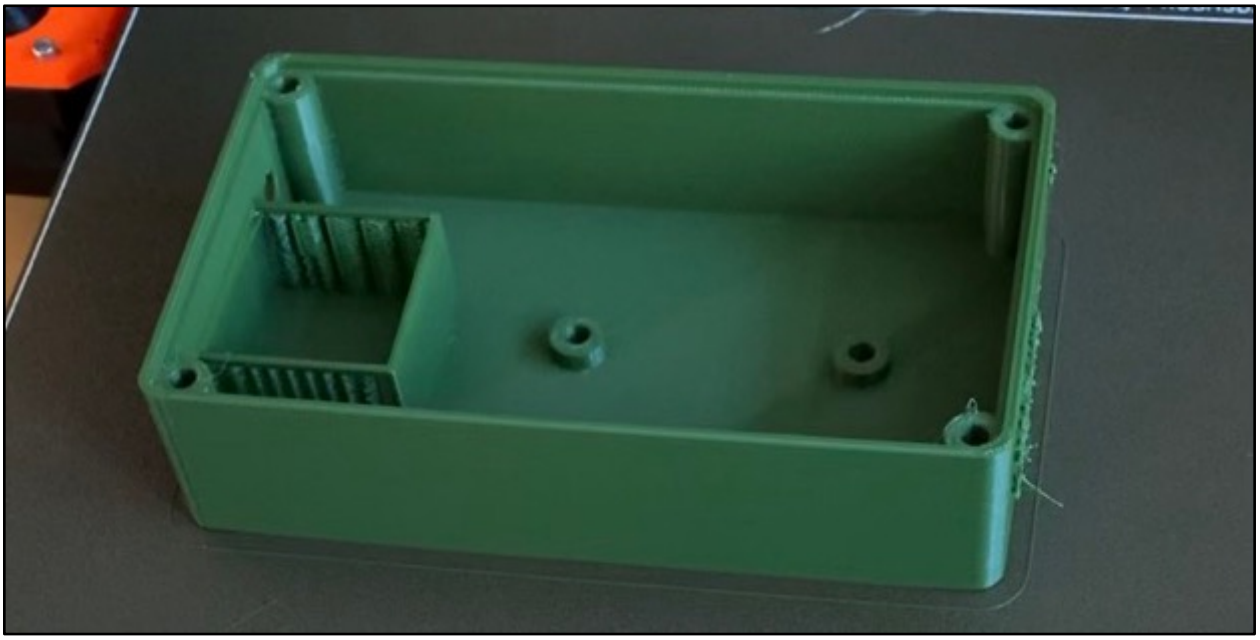
Project Video: https://youtu.be/oXSS2BYiA8w
Project GitHub: https://github.com/Liam-McCartney/Hardware
Desmos Rotation Visualization Graph: https://www.desmos.com/calculator/u5t4nidngs




Bread Board Prototype



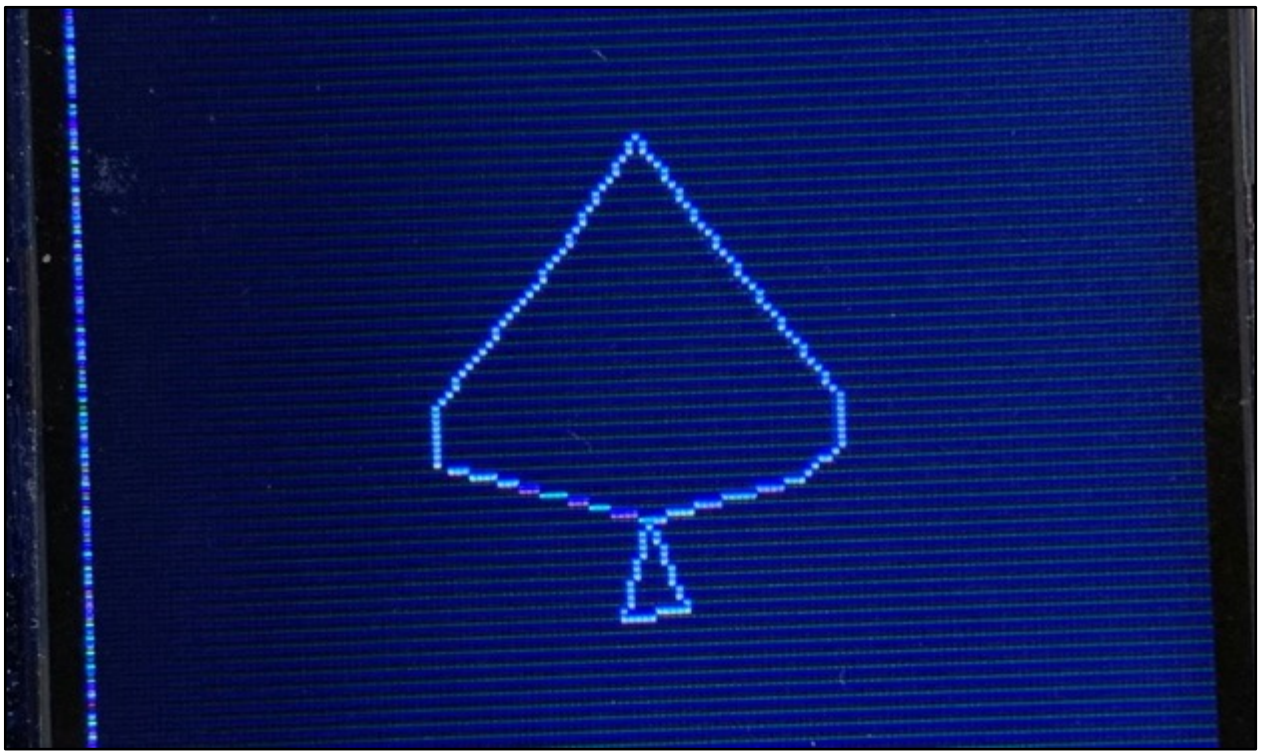
"Rat's Nest" (Internal Wiring)



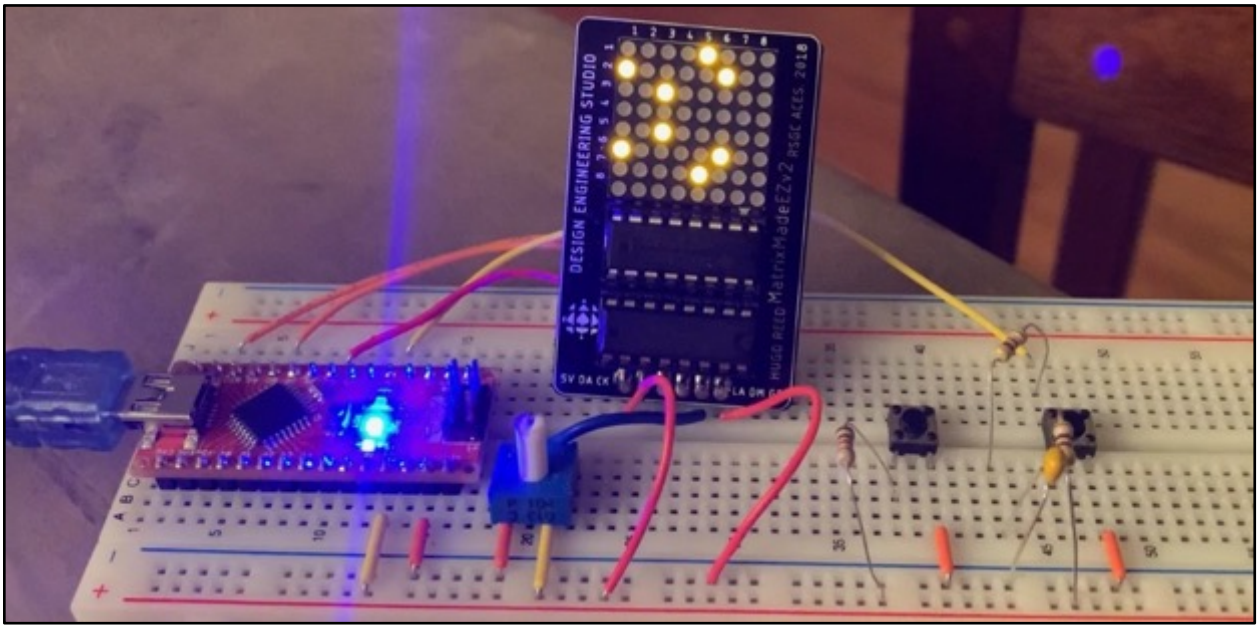
3D Printed Case Finishing Printing


First Successful LCD Test

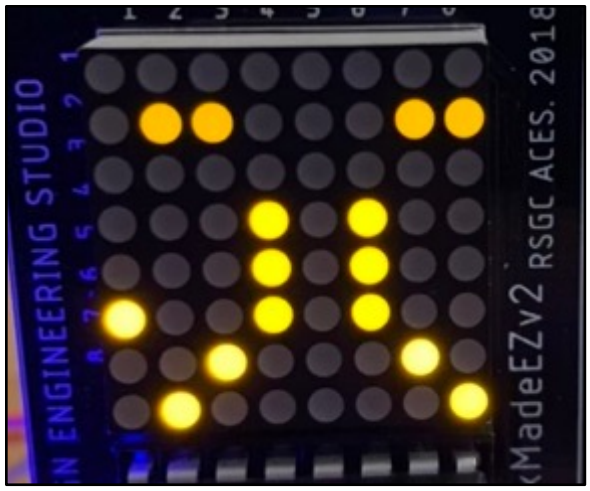
Pre-Optimization Half-Rendered Glitch



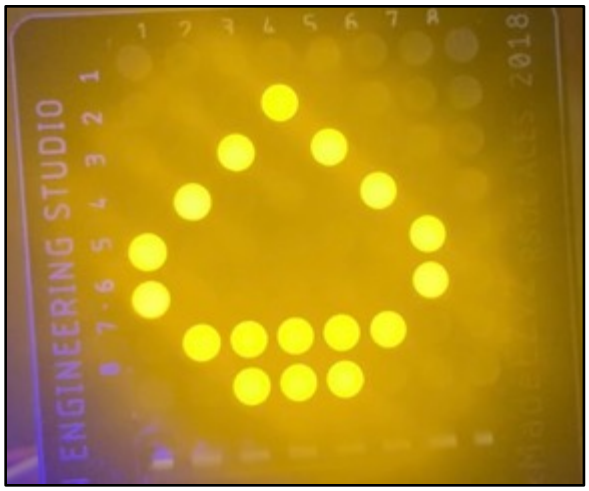
Poorly-Drawn 2D ACES Logo Rotating in 3D




Early WireFrame Rotator Prototype on LED Matrix




Early Struggles (Forgetting to Wipe Array)




ACES Logo on LED Matrix

## Code

Code Snippet:

```
// Project    : Medium ISP - The WireFrame Rotator
// Purpose    : Rotates 3D wireframes on an LCD display
// Course     : ICS3U
// Author     : Liam McCartney
// Date       : 2023 03 10
// MCU        : ATmega328p
// Status     : Working

// Full Code : https://github.com/Liam-McCartney/Hardware

void loop() {
  mapPoints();
  //Map 3D to 2D

  if (first) {
    //On the first itteration
    for (uint8_t i = 0; i < numLines; i++) {
      //Once for every line
      int8_t x1 = screenCoords[edgeTable[i][0]][0] * scale + 80;
      int8_t y1 = screenCoords[edgeTable[i][0]][1] * scale + 64;

      int8_t x2 = screenCoords[edgeTable[i][1]][0] * scale + 80;
      int8_t y2 = screenCoords[edgeTable[i][1]][1] * scale + 64;
      //Find line start and end coordinates

      TFTscreen.stroke(255, 255, 255);
      //Colour can be changed here

      TFTscreen.line(x1, y1, x2, y2);
      //Draw line
    }
  } else {
    //If it is not the first time
    for (uint8_t i = 0; i < numLines; i++) {
      //For every line...
      int8_t x1 = backupScreenCoords[edgeTable[i][0]][0] * scale + 80;
      int8_t y1 = backupScreenCoords[edgeTable[i][0]][1] * scale + 64;

      int8_t x2 = backupScreenCoords[edgeTable[i][1]][0] * scale + 80;
      int8_t y2 = backupScreenCoords[edgeTable[i][1]][1] * scale + 64;
```

```
      //Find the line between the coordinate's previous position

      TFTscreen.stroke(0, 0, 0);
      TFTscreen.line(x1, y1, x2, y2);
      //Paste over the line in black (erase)

      x1 = screenCoords[edgeTable[i][0]][0] * scale + 80;
      y1 = screenCoords[edgeTable[i][0]][1] * scale + 64;

      x2 = screenCoords[edgeTable[i][1]][0] * scale + 80;
      y2 = screenCoords[edgeTable[i][1]][1] * scale + 64;
      //Find the new coordinates of the line

      TFTscreen.stroke(255, 255, 255);
      TFTscreen.line(x1, y1, x2, y2);
      //Draw it
    }
  }
  delay(15);
  rotatePoints();

  for (uint8_t i = 0; i < numVertices; i++) {
    backupScreenCoords[i][0] = screenCoords[i][0];
    backupScreenCoords[i][1] = screenCoords[i][1];
    //Backup the points so that can be erased next cycle
  }

  if (first) {
    TFTscreen.background(0, 0, 0);
    first = false;
    //No longer first
  }
}
void rotatePoints() {
  for (uint8_t point = 0; point < numVertices; point++) {
    //Perform the rotation for every point using the rotation matrix
    float newX = vertices[point][x] * cosine - vertices[point][y] * sine;
    float newY = vertices[point][x] * sine + vertices[point][y] * cosine;

    //Update the coordinates
    vertices[point][x] = newX;
    vertices[point][y] = newY;
  }
}

void mapPoints() {
  for (uint8_t reps = 0; reps < numVertices; reps++) {
    //Once for every vertex
    float calcX = vertices[reps][x];
    float calcY = vertices[reps][y];
    float calcZ = vertices[reps][z];
    //Temporary storage variables for readability

    screenCoords[reps][x] = ((calcX * focalLength) / (focalLength + calcY));
    screenCoords[reps][y] = (focalLength * calcZ) / (focalLength + calcY);
    //Use formula to calculate the mapped coordinates
  }
}
```

## Reflection

This project was incredibly interesting for me. I found the exploration of dimensions and rotation to be incredibly deep rabbit holes. I had to stop myself from wasting time on an attempted rotating tesseract animation. That being said, I think I might change it up with the next ISP and move away from such a math focus. As for the timeline, it was quite a bit tighter than expected. Knowing I had a busy winter to come, I started this ISP back at the beginning of December. I believe I even had a working prototype on an LED matrix by the holidays. I actually had the rotation working on the LCD by early January and had optimized it by the end of the month. However, I did not budget enough for my now arch enemy: user interfaces. The incredibly simple UI took at least four times as long as I expected, and it was tedious. Even when it was done, there was no real sense of accomplishment since it was just a UI. On the bright side, I now know where I stand on the issue of UIs.

As for CAD and design in general, I saw incredibly improvement in this 3D print versus my previous case for the Data Logger. The Data Logger's case was big and bulky, but the WFR's case fits just right. In addition, the PCBs turned out well, minus the issue with the button wiring. The one thing about design that I struggled with was envisioning what the final product would look like. Up until I designed the case (about two weeks before presentations) I had no idea how every piece was going to fit together. However, in the end there was nothing to worry about as it turned out well.

I think I should also mention the learning that I had to do for this project. As someone who had never been exposed to the world of matrices and vectors before this project, it was a bit scary to solve my rotation problem with a rotation matrix. However, having now gone through it, I have learned two things. Firstly, I learned that there is a difference between knowing something and knowing how to program it. I had working rotation animations far before I understood the rotation matrix itself, let alone multiplying a vector by a matrix. It was a bit eye opening to how little you have to know to make it look like you know a lot, and I am not sure if that is a good or bad thing. Secondly, I learned that the world of linear algebra, while definitely complicated, is not as scary as it once seemed.

# Project 2.5a Mechanical

## Theory

The mechanical device of choice for project 2.5 is a homemade 5 V relay. Such a device operates using one of the four fundamental forces: electromagnetism. Electromagnetism governs the behaviour of electrons (and other charged particles) and electromagnetic fields. It is the reason that electricity works as it does, and includes phenomena such as magnetism. Relays are a type of switching device and rely on the laws of electromagnetism to create a magnetic field which triggers the device to switch. Relays are useful for their incredibly high switching voltage and current tolerance and their isolation abilities, which set them aside from other switching devices.

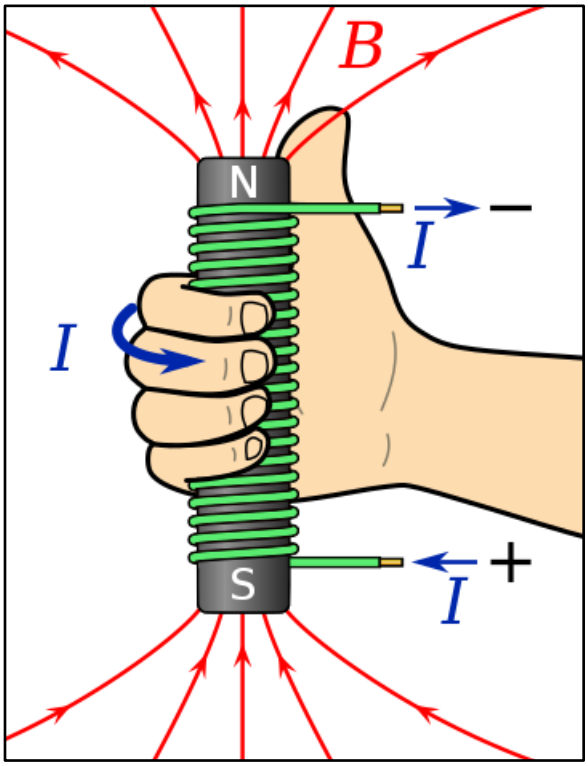
## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#mechanical

## Procedure

### Coils

As earlier mentioned, relays rely on electromagnetism to function. Relays are among a large group of mechanical devices that are controlled by copper coils, a type of inductor. Copper coils, and more generally inductors, are passive circuit components meant to turn changing electric fields into magnetic fields. This is possible due to one of the fundamental laws of magnetism: all changing electric fields create magnetic fields, and all changing magnetic fields create electric fields. By coiling copper wire, the small magnetic fields created by each point on the wire all point in the same direction, and therefore add together to create one big magnetic field following the right-hand rule. This magnetic field propagates from the coil, but is strongest at its core.



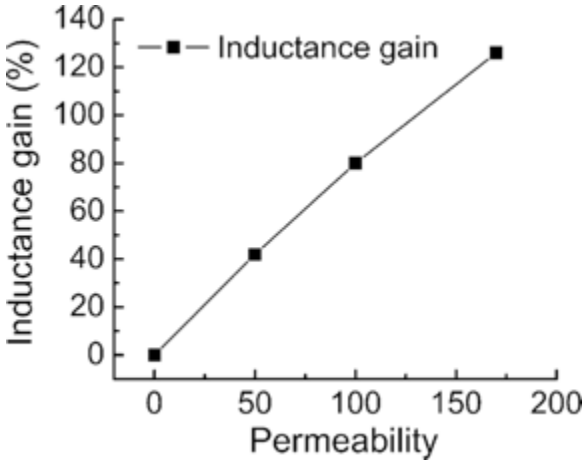

There are many different types of coils, but relays generally use solenoids. A solenoid refers to both a specific mechanical component as well as a coil wrapped around a cylindrical core, and both are good descriptions in this case. Relays use a solenoid shaped coil to create a mechanical solenoid to move physical connection pads in and out of contact, which is how relays complete their switching.

Solenoid coils have a relatively simple formula to calculate their inductance, which is a measure of the coil's strength in Henries:

$$L_{coil} = \frac{\mu_r \mu_0 N^2 \pi r^2}{l} = \frac{\mu_r \mu_0 N^2 A}{l}$$

$L$ is the inductance of the coil, $\mu_r$ is the relative permeability of the core, $\mu_0$ is the permeability of free space, $N$ is the number of turns, $r$ is the radius of the coil, $A$ is the area of the coil, and $l$ is the length of the coil. This equation is very useful for designing coils, but entirely useless without knowing what all the variables mean and the implications. Some are quite simple, such as more turns and a greater radius increases the inductance of the coil while having the same number of turns over a longer distance leads to a weaker coil. However, relative permeability is not so self-explanatory.

### Relative Permeability

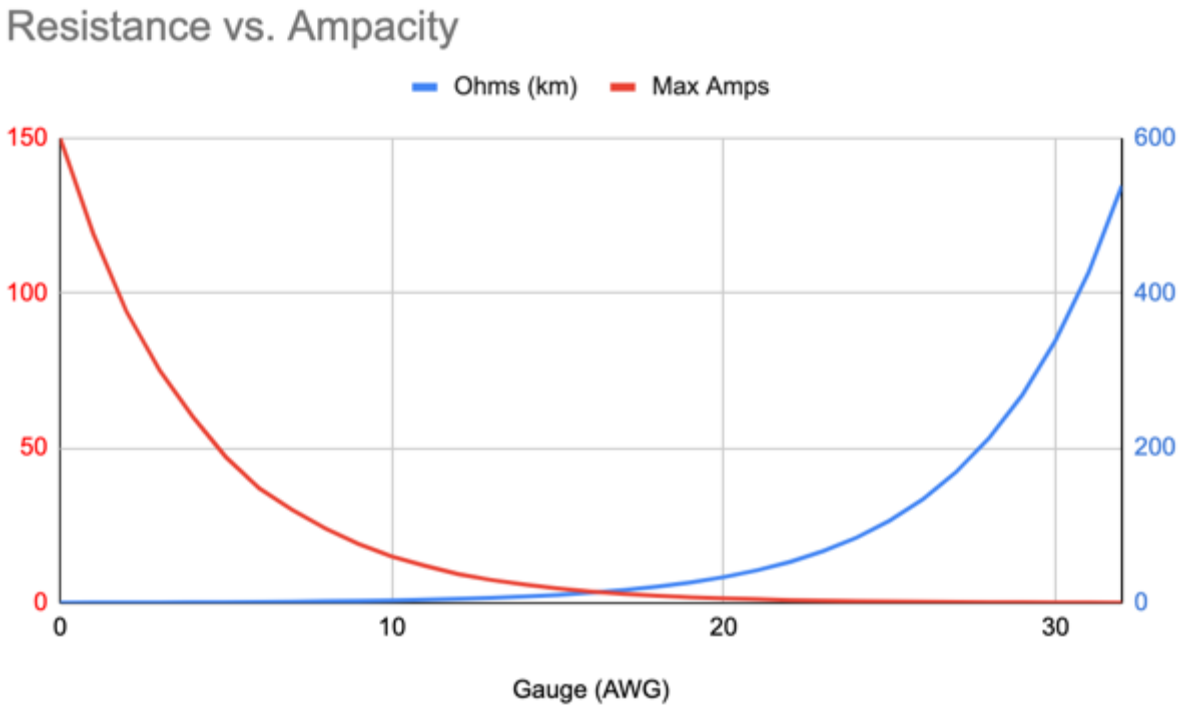



Permeability is perhaps the most important aspect of coil design. It is not so apparent when looking at the equation, but relative permeability is the reason that coils are feasible ways to harness magnetism. Relative permeability is a measure of inductance of a material. By using materials with high relative permeabilities such as iron or ferrite as the core to a coil, the magnetic field becomes more concentrated through the core and therefore gains strength. This gain is incredibly significant, as iron and ferrite generally have relative permeabilities near 100, which essentially means that using an iron/ferrite core instead of air will make an electromagnet roughly 100 times stronger. Due to this, purely air core coils are almost useless due to the low inductance compared to iron and ferrite core coils. Therefore, when designing a coil, it is of the utmost importance to maximize the amount of iron/ferrite contained in the coil in order to make it as efficient as possible.

The permeability of free space is a constant in physics, equal to $4\pi \times 10^{-7}$ Henries per metre. Free space is defined as a total vacuum, something that does not appear naturally on Earth. Air, which generally has a permeability closer to one Henry per metre, is a better comparison to grasp the scale of the increased performance a metal core gives a coil.

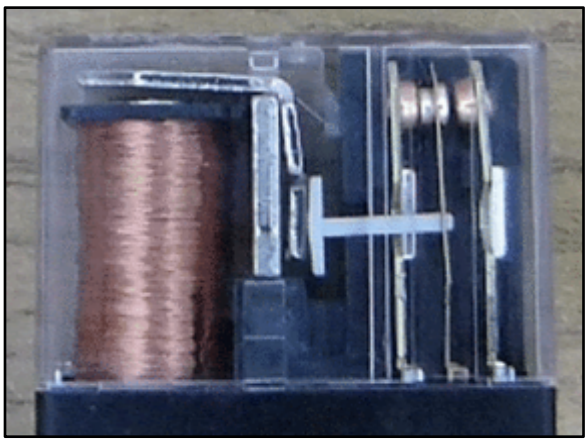
### Designing a Coil

While the topic of coils is a deep rabbit hole, their fundamentals are not too complicated. However, this changes when attempting to design one. Coils take lots of time and math to design before using, much unlike other components. For example, if you use the wrong wire or have too few turns, there could be a short circuit. Too thin wire could result in too high resistance and not enough current, or alternatively could melt due to too much current. The point is, coils take calculations before building them. The first step of designing a coil is to choose the wire. Almost all coils are made from enamelled copper wire, but the gauge of the wire is incredibly important. Thinner wire will have higher resistance per metre, but will have a lower maximum current capacity (called ampacity). On the other hand, thicker wire will have a high ampacity but low resistance. It is important to balance these two attributes, as a coil needs sufficient current to function but at the same time can quickly turn into a shorth circuit with too little resistance. On the next page is a graph that shows the relation of ampacity and resistance.




The general idea to stick to when designing a coil is to plan how long you want your coil to be and how many turns you want your coil to have, and then from there choose a wire that over that distance will provide enough resistance to prevent a short circuit. As long as your wire can handle the current of the circuit, the ampacity does not matter. That being said, it is good to stay as close to the maximum current of the circuit as possible as using a wire with too much resistance will begin to limit the current too much and the coil will be weaker than desired. Reliance on current is one of the main annoyances of coils. This feature prevents the use of series resistors to limit the current as they handicap the coil by providing a lower voltage drop across the coil and therefore less current. This means that the only solution to too little resistance is to add more turns.

## Relays



Conventional relays are mechanical devices that physically switch connections by moving the device. Solid state relays exist and are quite complex, but an original relay is relatively easy to create. All you need is a way to move your switching mechanism, and the mechanism itself. To the right is an example of a relay. The coil (on the left) is currently on, and therefore the metal arm is raised (reppeled by the magnetic field). This pulls the middle of the three contacts on the right towards the coil, creating a connection between the leftmost and centre contact pads. However, if the coil were instead attracting the arm, the connection would be between the rightmost and the centre pads. By setting the values of either the centre or outside pads, relays can control the states of the floating pin.

### Solenoids

As mentioned, the first component of a relay is something that can trigger the switching mechanism. This is generally a solenoid. Solenoids are simple devices; an iron/ferrite arm runs through the core of the coil with a magnet on the end. Powering the coil one way attracts the magnet, and the other repels the magnet. This is due to the different current directions and therefore opposite electromagnetic fields generated by the two modes. The use of a magnet on the end of the metal arm increases the strength of the coil for the same amount of current because the coil is attracted to the magnet as well as the magnet being attracted to the coil, essentially doubling the power. To the right is a picture of the solenoid used in this project. It is wrapped around a 3D printed cylinder with an iron arm. This makes its magnetic permeability significantly more than that of free space, though it is difficult to measure the exact value (estimated to be at a relative permeability of around 10-15). Under the 3D printed piece at the top of the coil is the magnet, attached to the arm. The entire solenoid is held in place by the relay stand, also 3D printed.

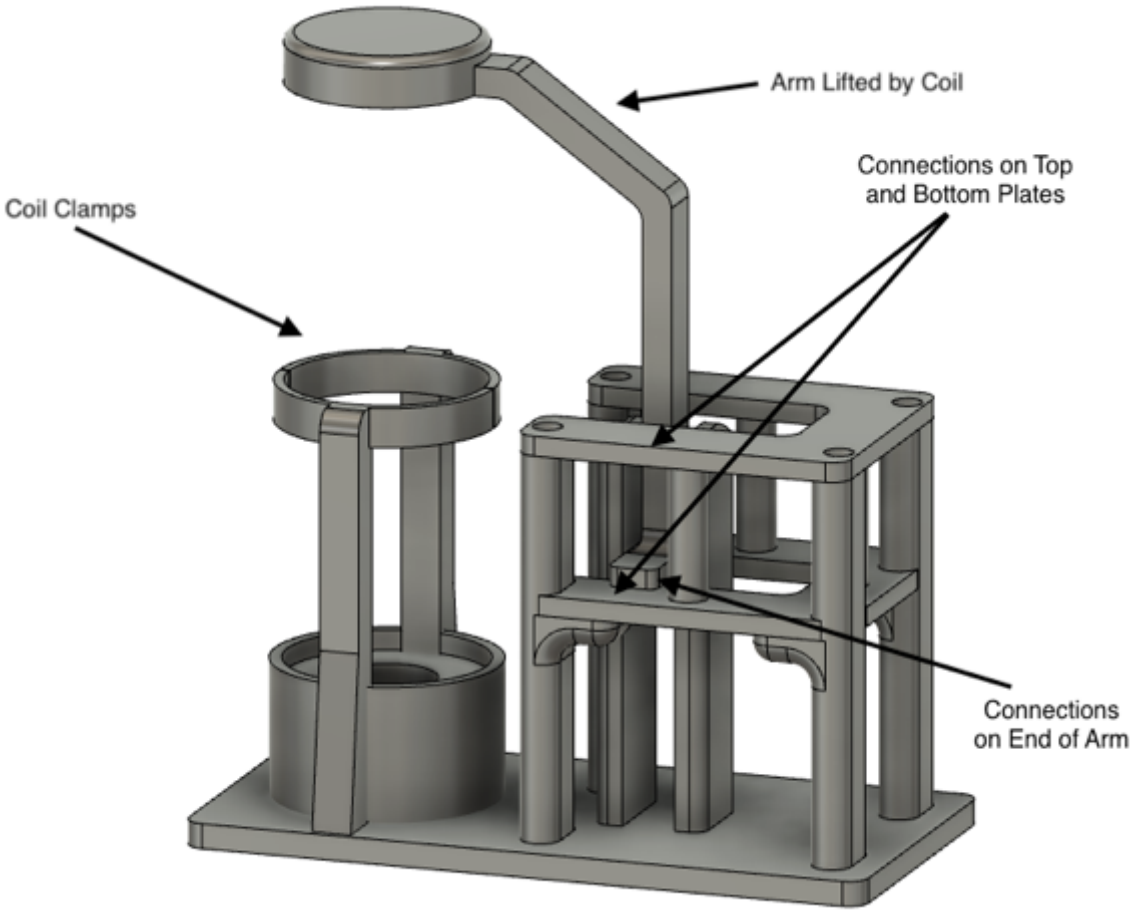
As mentioned, it is impossible to find the relative permeability of the coil's core with household tools, but it is possible to find a range inductance. Using the formula from earlier, we can find the inductance of the coil. It has roughly 600 turns, a length of 5 cm, and an average radius of 15 mm. Note the units for all lengths is metres.

$$L_{coil} = \frac{\mu_r \mu_0 600^2 \pi 0.015^2}{0.05}$$

Therefore, using the earlier assumption of relative permeability, we find the inductance of the coil ranges from 0.0064 Henries to 0.0096 Henries. While this value may seem quite small, Henries are quite a large unit compared to other SI units. While most of the base units such as a single metre or a single volt may not be considered a lot, a single Henry is a very respectable amount of inductance. Interestingly, this stems from its definition, found in Faraday's famous equations that say that changes in magnetic field induce voltage in conductors. A Henry is defined as the necessary inductance to induce one volt of potential into a conductor, which happens to actually be quite a high amount of inductance. Most inductors have an inductance far off of even a single Henry.

### Switching Mechanism

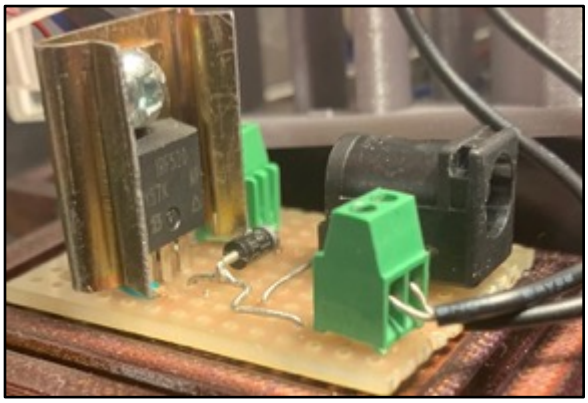
The next task at hand to build a relay is to somehow translate the movement of the solenoid into the separation and connection of contacts. Traditionally this is done similarly to the earlier example by translating vertical motion into horizontal motion, but this approach is overly complicated for a homemade relay and is difficult to manufacture. Instead, it is simpler the keep all motion in one direction, in this case vertical. On the next page is an image of the stand made for this relay, where the switching mechanism can be seen.




As seen in the diagram above, the switching of the solenoid relies on the arm, which is raised and lowered by the coil. When lowered, the arm connects with the top of the bottom piece of the standing structure, which is one throw of the switch. When raised, the arm connects with the other plate, creating the other possible connection. The whole apparatus is 3D printed in four parts: the arm, the main stand, the top connection plate, and the connection bottom plate. The last two must be printed as their own parts to avoid overhang.

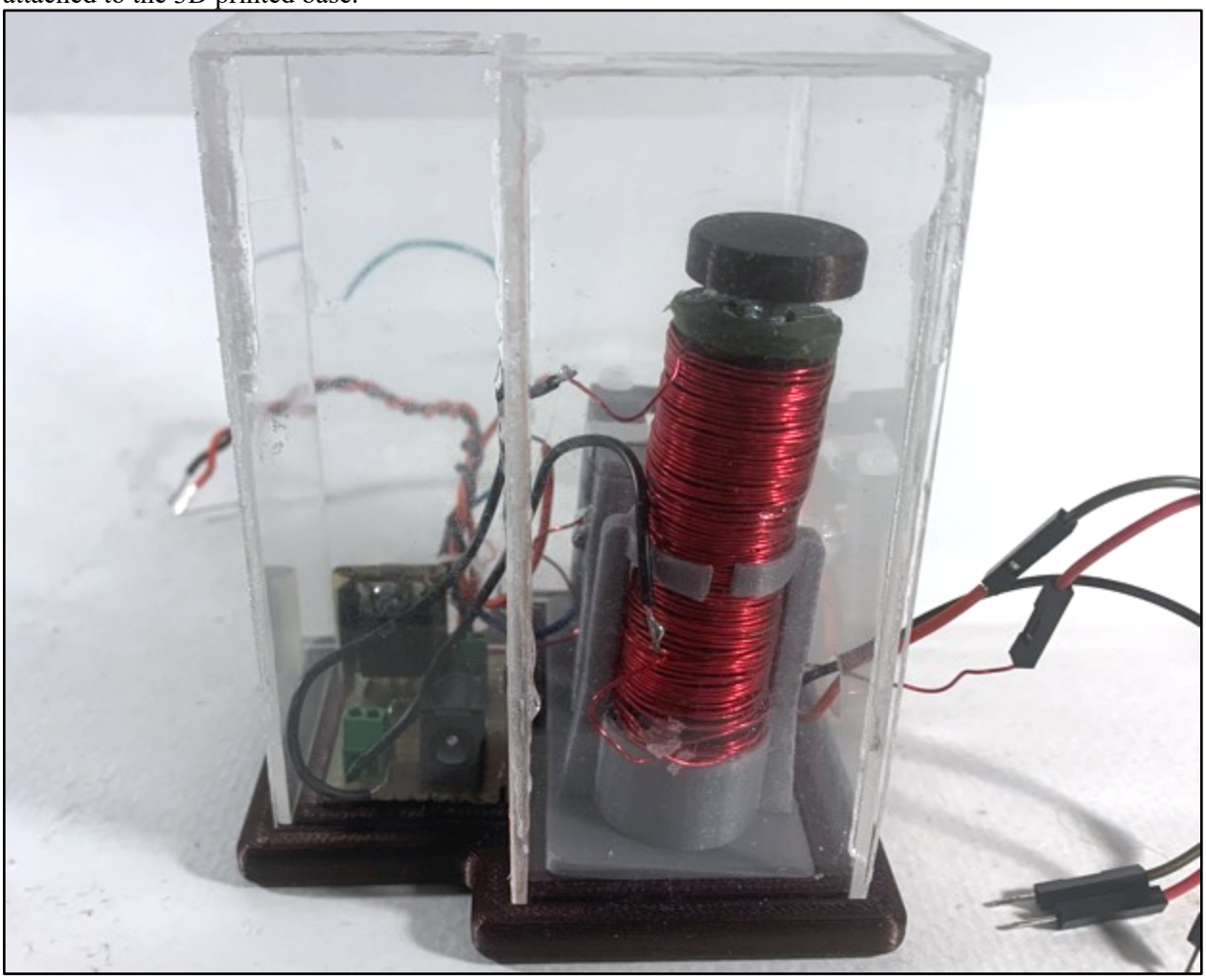
## Hardware & Assembly

While this coil has over 600 turns of wire and a relatively high resistance wire, the total resistance across the coil is only 2.2 Ω. This results in a current draw of roughly 2.27 amps according to Ohm's law, but in practice about 1.6 amps as the heat of the coil increases its resistance, and the coil heats up quite quickly at 2.27 amps. This means that powering the coil with a microcontroller directly is out of the question, and even BJT transistors cannot source (or drain) enough current to switch the coil. Therefore, somewhat ironically, a MOSFET is needed to switch the coil.



To the right is an image of the switching board for the relay, which also includes a flyback diode to protect from voltage spikes from the collapsing electromagnetic field of the relay. There is a barrel jack on the board, but through practice it seems that the connection through the barrel jack drops roughly 0.4 amps of current, most likely due to resistance with the connection. It works, but not with 5 V of power and the 9 V power adapters from the kits cannot source over 1 amp of current, so the barrel jack is essentially useless for the time being. This board works by allowing current to go to ground though the MOSFET (low-side switching) when the input to the board is high and blocks current when the input is low. While the IRF520 MOSFET is more than capable of sinking currents around 1.6 amps, it does heat up very quickly (burning temperatures in less than 10 seconds). Thus, a make-shift heatsink was added and fixes the problem with the MOSFET never reaching dangerous temperatures.

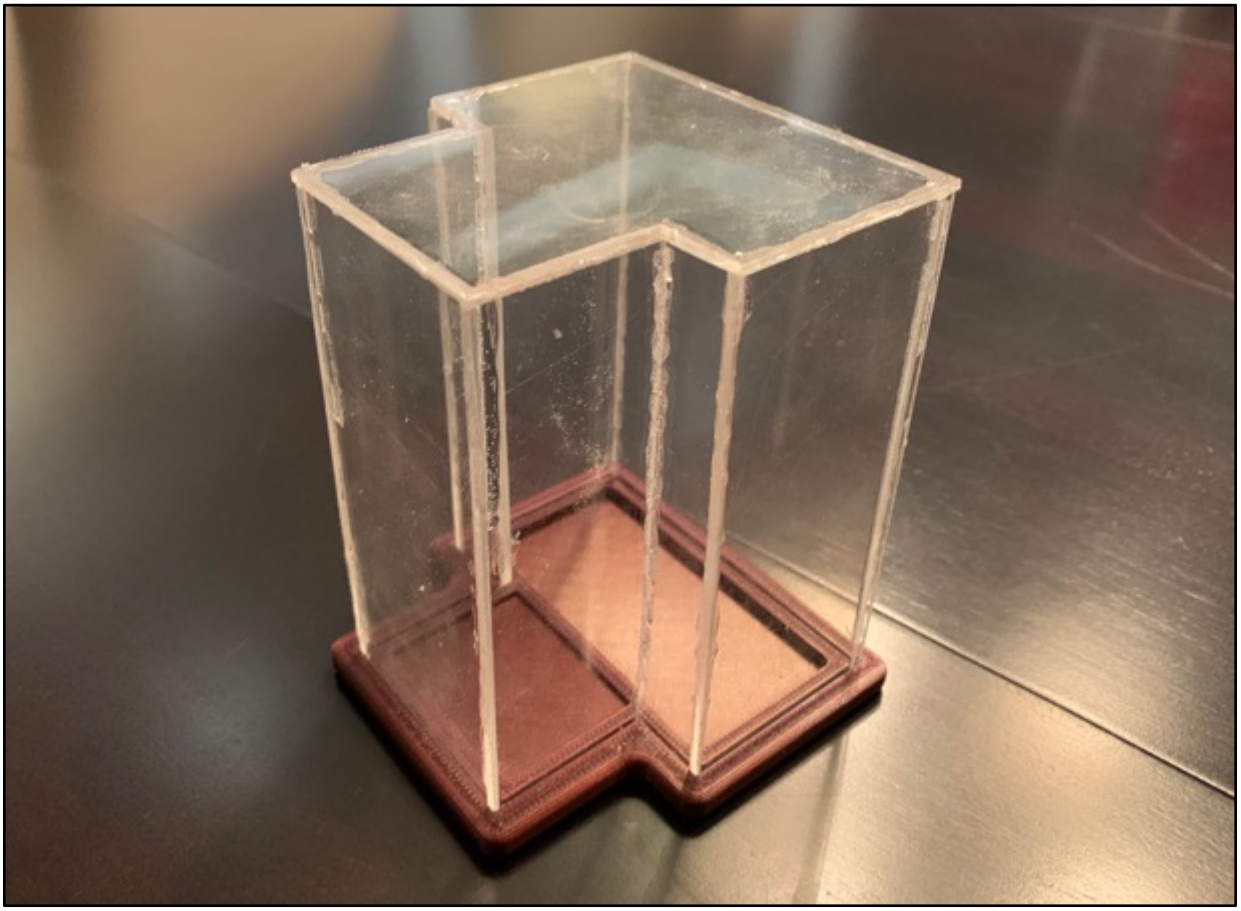
In addition, an encasement for the entire relay and switching board was created out of acrylic with a 3D printed base, as shown below. The acrylic panels are joined to one another by plastic cement but are not attached to the 3D printed base.



| Parts Table | |
|---|---|
| **Relay Stand & Case** | N/A |
| 3D Printed Relay Stand | 1 |
| 3D Printed Base | 1 |
| Acrylic Panel | ** |
| Aluminium Tape | ** |
| M3 Screw | 4 |
| **Solenoid** | N/A |
| 3D Printed Hollow Core | 1 |
| Iron Screw | 1 |
| Fridge Magnet | 1 |
| Magnet Wire (Enamelled Copper Wire) | ** |
| **Switching Board** | N/A |
| IRF520 MOSFET | 1 |
| Schottky (Flyback) Diode | 1 |
| Terminal Block | 2 |
| Strip Board | 1 |

## Media

Project Video: https://youtu.be/MgbWF_15q8M
Fusion Files: https://github.com/Liam-McCartney/Hardware/tree/main/Mechanical%20(Part%201)



Empty Acrylic Case

## Reflection

I found this project very interesting and enjoyable, and it came at the perfect time. Prior to the beginning of the mechanical unit in class, I had been preparing to make some coil devices on my own time. While I definitely was not intending to make a relay (I actually wanted to make a small slayer exciter), I had already interested myself in the world of coils. As soon as I saw the green text in the project description which said that coils were an option, I set my sights on a homemade mechanical device. I originally thought about making a solenoid, but quickly realized that it would be too easy as it is essentially just wrapping lots of wire. Naturally, the next step was a relay, as they incorporate solenoids.

Through designing my relay and building it I learned a lot of useful skills, mostly related to 3D design. The switching mechanism I made for this project is multiple times more intricate than any other thing I had designed before, incorporating my first moving parts. I am quite happy with how it turned out, though I did need to reprint one part. When I finally saw my project work after hours of design, I felt such an achievement. For this project we were pushed to improve our design skills, and not only did I do that but I also began to see the importance of design, and the creative power of being able to bring anything to life with Fusion 360 and a 3D printer.

Lastly, in regards to time management, I was able to complete this project slightly faster than anticipated and I did not end up needing my one-day extension, even with very limited time on Saturday.

# Project 2.5b Wireless Control

## Theory

This project employs the basics of radio communication and radio frequencies (RF) to achieve wireless communication between two microcontrollers using only two resonant tank circuits. In doing so, this project teaches about many principles, such as radio waves, electrical resonance, and communication protocols.

## References

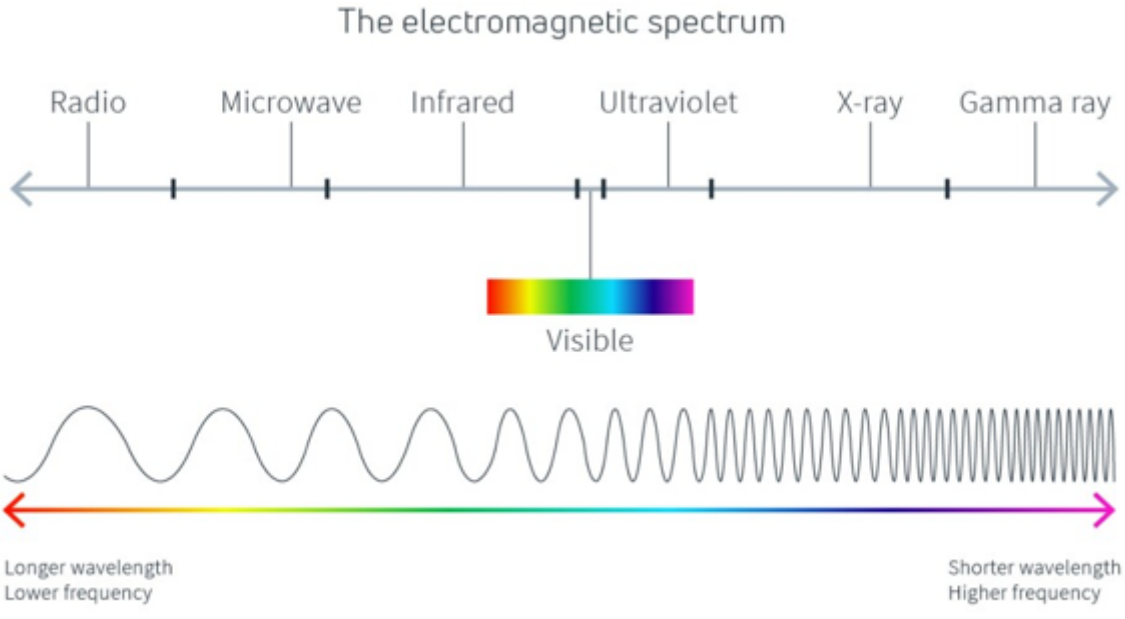
Project Description: http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#wireless
AM Radio: https://www.youtube.com/watch?v=mcD3uBld4Y4&t=491s&ab_channel=HyperspacePirate
LC Calculator: https://goodcalculators.com/resonant-frequency-calculator/
LC Circuits: https://en.wikipedia.org/wiki/LC_circuit

## Procedure

This project is naturally broken into two stages: the actual RF transmitter and receiver (as well as their theory), and the communication method used to engage the possibilities of the transmitters and receivers.

### Radio Frequencies & Communication

The first thing to identify is the meaning of a radio frequency. A radio frequency is generally agreed to be any frequency between 3 kHz and 300 GHz, though lower and higher frequencies are sometimes used in extreme applications. This means that the vast majority of frequencies used for communication around the world fall under the range of RF, including Wi-Fi and cell-phones.

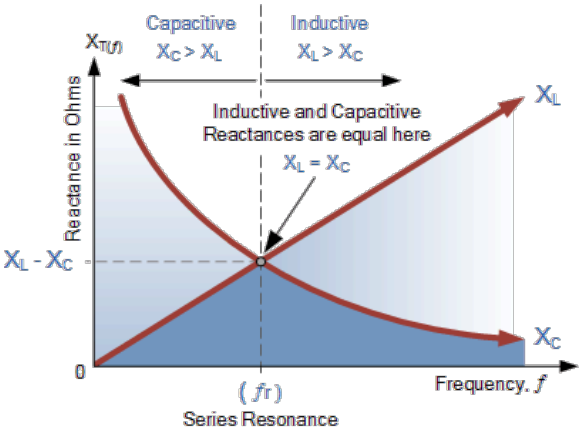



However, the frequency alone does not make radio transmission. Radios transmit data through radio waves, which are a type of electromagnetic wave (EMW). Radio waves are emitted when conductive materials are subjected to an electric signal oscillating at a radio frequency. To receive data, antennas absorb these EMWs created by the transmitter, inducing a voltage into the receiver. This voltage can then be measured and turned into useful information.

In general, higher frequency EMWs will travel shorter distances and are more easily disrupted than lower frequencies, but can carry data at higher rates than lower frequency transmissions, which travel further distances. Despite this, frequency alone does not determine range. Range is foremost dependant on the amplitude of the signal generated by the antenna, where amplitude is the voltage difference between the peaks/troughs and average of the waveform. Amplitude can be affected by a few things, such as antennas, amplifiers, and sensitivity, but in general the most effective way is to set the resonant frequencies of the transmitter and receivers to be the same as the carrier frequency (the frequency being broadcasted).

### Impedance & Resonant Tank Circuits

One way to set the specific resonance of a circuit is to use a resonant tank circuit, or an LC circuit. Before explaining how they work, it is important to note that the electrical definition of resonance is different than the mechanical definition. When an electrical circuit is at resonance, the amplitude of the circuit's voltage over time is maximized. This leads to and inherent amplification of the carrier signal, assuming it is the resonant frequency, and naturally filters out other background frequencies.



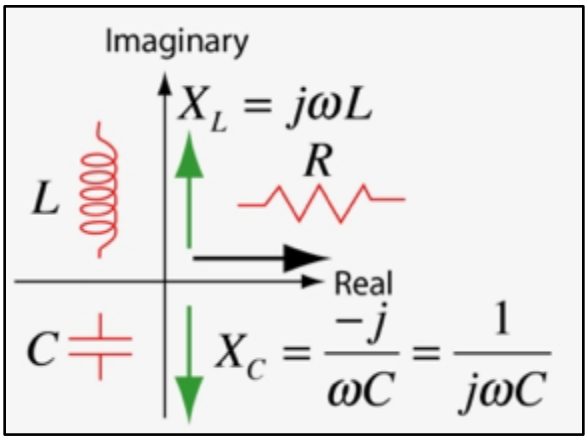

Electrical resonance has very little to do with the conventional definition of resonance, and this is because electrical resonance is actually a function of impedance, not anything to do with wave length and frequency. The simplest definition of a circuit at resonance is the when the resistance experienced by the load is purely real. By this definition, a simple DC circuit with a resistor is at resonance. However, in AC circuits with inductors and capacitors, the circuit is at resonance when the capacitive reactance ($X_C$) and inductive reactance ($X_L$) have equal magnitudes, meaning the imaginary components of the resistance cancel and the resulting resistance once again is purely real. However, since reactance is a function of frequency, the impedance will have an imaginary component at all but one frequency. That one frequency is named the resonant frequency.

Now that the explanation is done, it is time to understand. A lot of terms were thrown around in the previous paragraph, and it is now time to sort them out, beginning with reactance. With DC, or an AC circuit oscillating at 0 Hz, inductors act as short circuits and capacitors act as open switches, besides for both component's Ohmic resistance. However, when the frequency begins to increase, capacitors begin to act more like ideal conductors while inductors begin to act as resistors. This is because voltage across a capacitor cannot change instantaneously, and since the capacitor is not given sufficient time to charge or discharge at high frequencies, it simply acts as if it allows current to pass through with no resistance (at infinity). On the other hand, inductors inherently resist changes in current, as described by Faraday's and Lenz's. This function of inductors is due to their produced EMF, which is resistive to any change, thus resisting the change of current direction from AC.

The resistance that these two components place on AC signals are both called reactance (though with different notation, $X_C$ and $X_L$), and can be calculated using their respective formulas. Reactance is measured in Ohms, but it is important to note that reactance is an imaginary measurement, which will be important later.

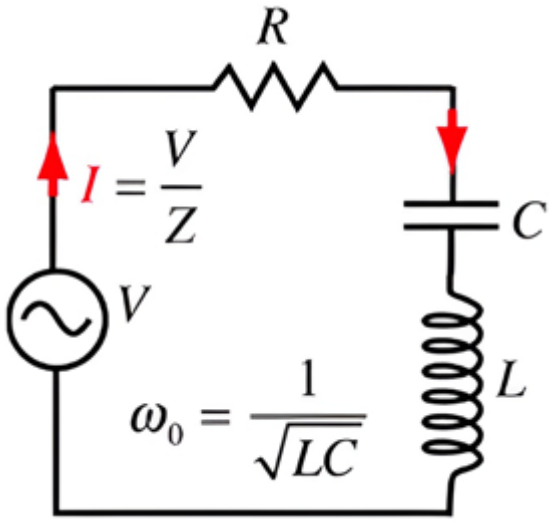
$$\text{Capacitive Reactance: } X_C = \frac{1}{2\pi f C} \qquad \text{Inductive Reactance: } X_L = 2\pi f L$$

With these two measurements, it is time to talk about impedance. Impedance is essentially a measurement of AC resistance, including both real and imaginary components. Impedance is measured in Ohms, and has an angle, which tells whether or not the impedance is real, and if not whether it is capacitive or inductive. Impedance can be shown on a 2D number line with a real axis and imaginary axis, where the imaginary axis has inductive reactance on one side and capacitive on the other. The impedance is then a plot on the resulting plane, making sure that inductive reactance and capacitive reactance cancel out. Thus, impedance is actually a complex measurement.




With the understanding complete, a resonant tank circuit is a simple circuit that consists of a capacitor and an inductor in series (resistance does not affect the resonant frequency). Resonant tank circuits are used when a circuit needs a specific resonant frequency.

At resonance, the reactance of the inductor and capacitor will cancel out, reducing the circuit's resistance to only the real measurement of any load, plus the real resistance of the capacitor and inductor. Since the magnitude of the reactance of both the capacitor and inductor will be equal in the case of resonance, we can build an equation to find the resonant frequency of any resonant tank circuit. This equation also has any ramifications, such as being able to find accurate inductance using a known capacitor.




To build this equation, we simply start with the knowledge that both the capacitive reactance and inductive reactance equations will be equivalent at the resonant frequency.

$$2\pi fL = \frac{1}{2\pi fC}$$

From there, it is simple enough to isolate for $f$ and arrive at the equation for the resonant frequency of a resonant tank circuit:

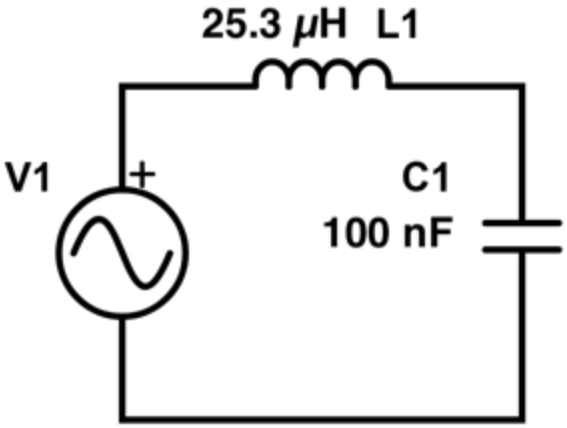
$$f_{resonant} = \frac{1}{2\pi\sqrt{LC}}$$

By picking inductor and capacitor values that create a circuit that resonates at the frequency of the carrier signal, you can greatly improve the range and signal strength of RF communication. In practice, it is best to start with a given capacitance and find the inductor that will satisfy the equation, and then wind one by hand. To check the inductance, you can use an oscilloscope and signal generator to make sure the amplitude is the highest at the desired frequency.

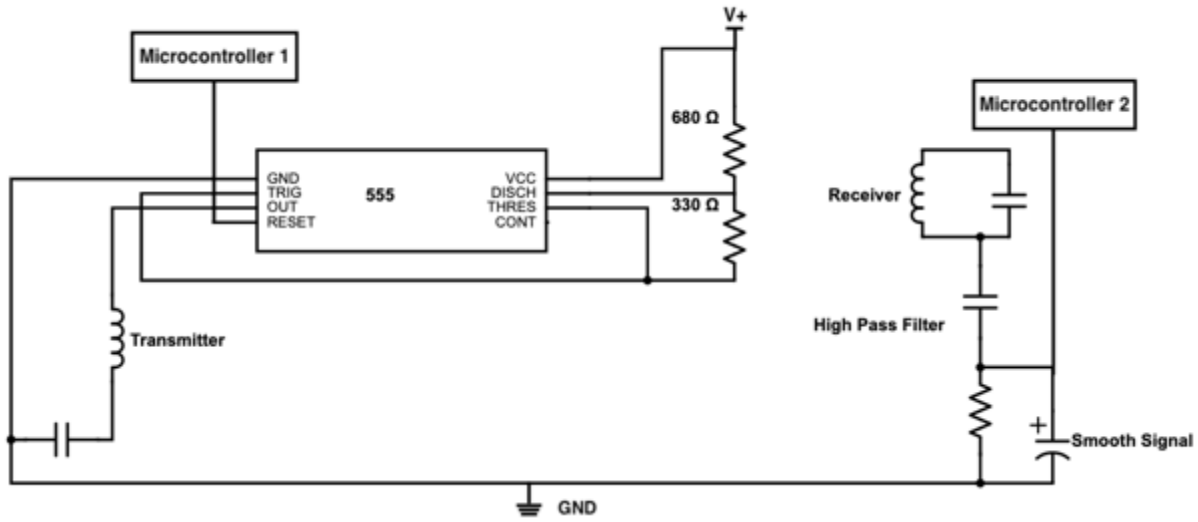
## Specific Circuit

The frequency used for this project was 100 kHz in the form of a square wave generated by a 555 timer. This setup was chosen as the 555 timer has a formula to determine the frequency of the square wave generated, so in theory it would be relatively easy to get exactly 100 kHz (in practice, the formula, which is only an approximation, seemed to break down at frequencies well above the normal use of such a timer). This meant that the antennas had to be configured to resonate at 100 kHz, since in this project the antennas were the coils themselves. Using a real antenna results in longer ranges, but it is incredibly hard to calculate for its resonant frequency as it relies on the capacitance of the antenna as well as parasitic capacitance through the air.

Using the formula from earlier, a predetermined 100 nF capacitor resonating at 100 kHz has a reactance equal to that of a 25.3 µH inductor. To actually make an inductor of that measure requires 60 turns in 5 cm with a diameter of 0.6 cm. One interesting thing is that inductance is technically a rate change of turns per root of the length. Either way, the resulting resonant tank circuit to with values is shown on the right. Note how the capacitor is non-polar in the schematic. While not necessary in some scenarios, non-polar capacitors provide much less hassle when creating resonant tank circuits. As 100 nF ceramic capacitors are not common components, 10 10 nF capacitors were used in parallel.

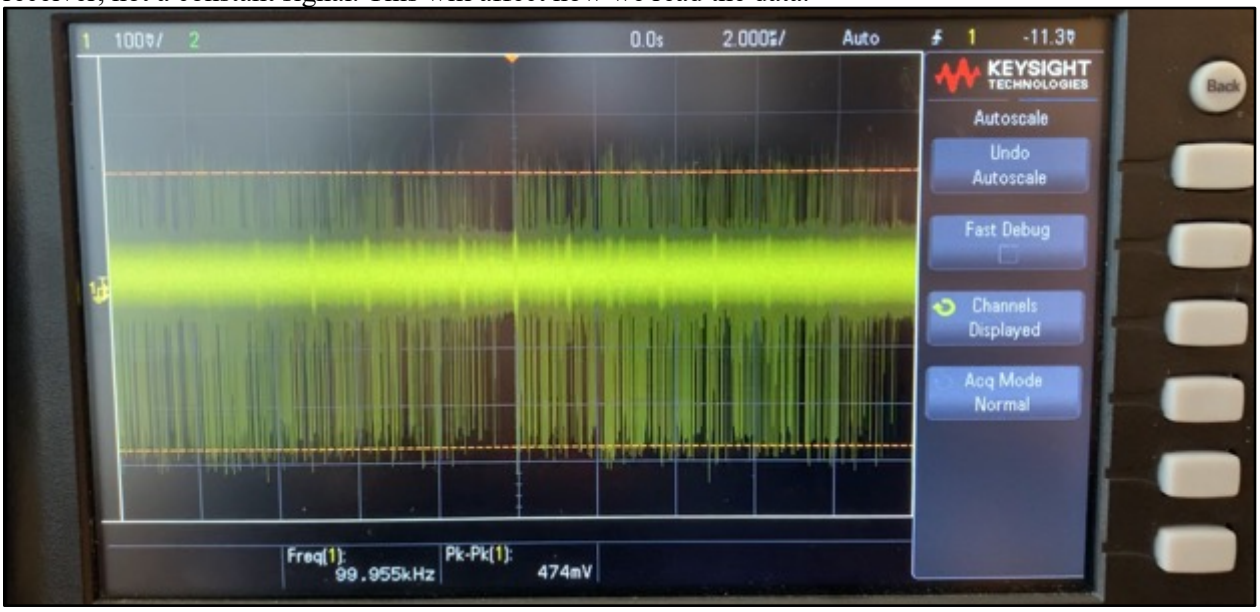



Below is a schematic of the entire circuit used. Note the common ground connection. Common ground is a good way to improve the range and connection strength of RF communication, and is employed in the real world. For example, cell-phone towers are all connected to Earth ground through either ground rods or ground sheets. In this project, a wire was used to connect the receiver and transmitter to a common ground, but this is not necessary if both are connected individually to Earth ground, which would make the setup completely wireless. Common ground provides a more accurate reference for the receiver circuit, making it more sensitive to incoming EMWs specifically from the transmitter.



## Communication Protocol

Now that we are able to transmit signals over radio waves, it is time to come up with a communication protocol to transfer these waves into useable data. Before doing that, it is essential to understand the nature of the signal being received and shown to the receiver microcontroller. Below is a picture of an oscilloscope reading of the signal, which is actually just a series of short pulses of induced voltage in the receiver, not a constant signal. This will affect how we read the data.



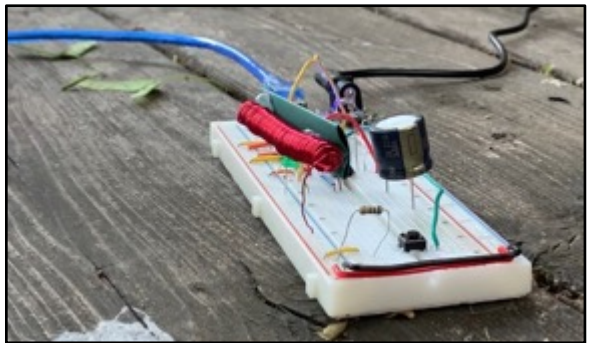

This signal is controlled by the transmitting microcontroller by pulling the reset pin of the 555-timer high when it wants to send a signal, and low when it does not want to send a signal. The chain of events for sending a bit is as follows: First, send a 50 ms signal. Then, for 150 ms, sent the bit. If the bit is high, this means sending a signal, and if it is low, not sending a signal. Then delay for 100 ms to give possible time for the receiver to catch up.

This makes the receiving algorithm quite simple as well. First, listen for the 50 ms signal which alerts the receiver that a bit is coming in. When the receiver hears it, it delays 50 ms to wait for the bit. Then, if the receiver reads another signal within a 150 ms window, the bit is high, and if the receiver reads nothing, the bit is low. These timings are very long which does lead to quite slow communication. As a result, there is no threat of bits being lost in transmission, as handling that issue would be a new challenge unrelated to basic RF communication.

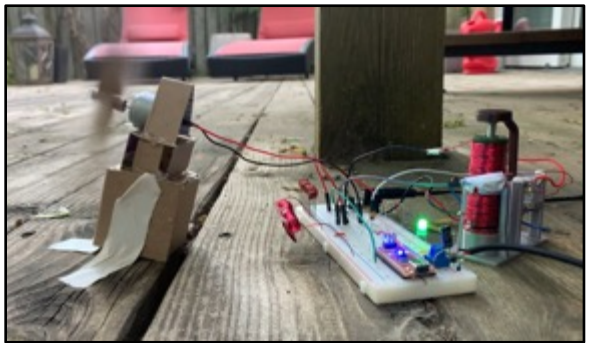
## Final Product

As mentioned, the final product uses a common ground connection for increased range. This is because earlier tests that did not have common grounds had a maximum range of less than 20 cm, which is disappointing. To the right is a picture of some early testing, which taught the value of common ground, which when implemented not only allowed longer range but also better connectivity.

It is this common ground connection that is the reason for the wire connecting the receiver and transmitter that can partially be seen in the picture below. However, the only reason that this trade-off was accepted is that a common ground does not theoretically require a wired connection. Once again, connecting each circuit to Earth ground would eliminate the need for the wire seen below, but to do so requires the proper equipment. Below is a view of the transmitter and receiver.



Transmitter



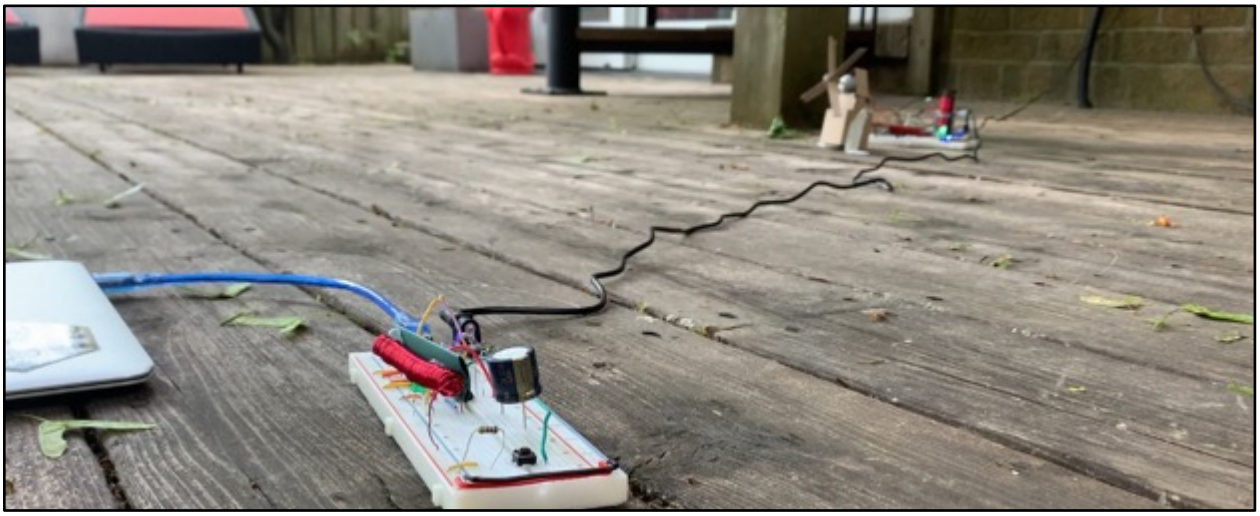
Receiver

The final design for this project is a fan powered by the relay from the previous project. This fan is simply a motor with blades. Due to their current draw, the motor and relay share a power supply separate from the rest of the circuit.

| **Parts Table** | |
|---|---|
| **Transmitter** | N/A |
| Arduino Nano | 1 |
| Custom Wound Coil | 1 |
| Button (PBNO) | 1 |
| 555 Timer | 1 |
| Assorted Resistors | 4 |
| Assorted Wires | ** |
| **Receiver** | **N/A** |
| Custom Wound Coil | 1 |
| 10 nF Ceramic Capacitor | 10 |
| Arduino Nano | 1 |
| 10 kΩ Potentiometer | 1 |
| 330 Ω Resistor | 1 |
| 1 µF Polarized Capacitor | 1 |
| Greed LED | 1 |

## Media

Project Video: https://youtu.be/zwFE9EbbuoU
Project GitHub: https://github.com/Liam-McCartney/Hardware/tree/main/Wireless



Wide View of Transmitter and Receiver

## Code

Transmitter Code:

```
// Project   : Wireless Communication
// Purpose   : Transmits data to receiver
// Course    : ICS3U
// Author    : Liam McCartney
// Date      : 2023 05 29
// MCU       : 328P (Nano)
// Status    : Working
// Reference : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#wireless

#define comPin 2 //Pin used to power coil through 555

void setup() {
  Serial.begin(9600);
  while (!Serial)
    ;
  Serial.println("-----------");
  Serial.println("Hello World");
  pinMode(comPin, OUTPUT);
  digitalWrite(comPin, 0);
}

void loop() {
  if (digitalRead(3)) digitalRead(A5) ? sendByte(0b11111111) : sendByte(0b00000000);
  //Sends either on or off cod depending on switch position
}
/**************
Protocol:
1. Send 50 ms HIGH to alert receiver
2. After 50 ms HIGH, send bit for 150 ms
3. Delay for receiver to catch up if needed
**************/
```

```
void sendBit(uint8_t bit) {
  if (bit == 0) {
    Serial.println("Sending 0");
    digitalWrite(comPin, 1);
    delay(50);
    digitalWrite(comPin, 0);
    delay(250);
  } else {
    Serial.println("Sending 1");
    digitalWrite(comPin, 1);
    delay(50);
    digitalWrite(comPin, 1);
    delay(150);
    digitalWrite(comPin, 0);
    delay(100);
  }
}

void sendByte(uint8_t mybyte) {
  Serial.print("Sending Byte: ");
  Serial.println(mybyte);
  uint16_t mask = 1;

  for (int i = 0; i < 8; i++) {
    uint8_t bit = mybyte & mask;
    mask = mask << 1;
    sendBit(bit);
    //Send the byte bit by bit using a mask
  }
}
```

Receiver Code:

```
// Project   : Wireless Communication
// Purpose   : Receives data from transmitter
// Course    : ICS3U
// Author    : Liam McCartney
// Date      : 2023 05 29
// MCU       : 328P (Nano)
// Status    : Working
// Reference : http://darcy.rsgc.on.ca/ACES/TEI3M/2223/Tasks.html#wireless

#define recPin A0  //Input pin
#define outPin 2   //Output pin

uint8_t thresh = 0;
//Reading is only used if it is over the threshold, filters noise

#define threshPin A2
#define threshHigh A3
#define threshLow A1

void setup() {
  Serial.begin(9600);
  while (!Serial)
    ;
  Serial.println("-----------");
  Serial.println("Hello World");

  pinMode(outPin, OUTPUT);
  digitalWrite(outPin, 0);
```

```
  pinMode(threshHigh, OUTPUT);
  pinMode(threshLow, OUTPUT);

  digitalWrite(threshHigh, 1);  //Pot pins
  digitalWrite(threshLow, 0);
}

//-------------------------------------//
//Variables for loop():
uint32_t currentTime = 0;
bool proceed = false;

uint8_t mybyte = 0b0;
uint8_t bit = 0;

bool sign;
//-------------------------------------//

void loop() {
  thresh = analogRead(threshPin) >> 6;
  //Find the threshold

  if (analogRead(recPin) > thresh) {
    //Interrupt impossible as analogRead() is needed

    delay(75);
    //If signal is detected, wait for initial alert to end

    currentTime = millis();
    while (100 > millis() - currentTime) {
      //For exactly 100 ms check the bit after the alert
      if (analogRead(recPin) > thresh) {
        sign = true;
      }
    }

    delay(50);
    //Delay to prevent overlap

    Serial.println("---------");
    Serial.println("Received!");
    if (sign) {
      //If the bit is HIGH...
      mybyte = mybyte | (1 << bit);
      sign = false;
    }
    ++bit;
    //Next bit
    Serial.println(mybyte);

    if (bit == 8) {
      //If byte has been transmitted...
      bit = 0;
      Serial.println("Setting LED");
      mybyte ? digitalWrite(outPin, 1) : digitalWrite(outPin, 0);
      //Turns the relay off or on depending on the code sent
      mybyte = 0;
      //Reset byte
    }
  }
}
```

## Reflection

I am going to keep the reflection for specifically this project pretty short. Overall, I am proud of what I accomplished with it, going from knowing very little about RF to more than the average person in a short period of time, as well as developing my own transmitter/receiver circuit and communication protocol. While the common ground wire that I used definitely took away from the grandeur of the project, I am still happy with how it turned out. I tried hard with the completely unconnected circuits, but could simply not accept the short range and poor connectivity.

Now, moving onto my reflection for the year. I have learned so much this year in hardware, and truly I did not even realise it until recently as I have begun to look back on grade 11. This week I found the first sketch that I wrote the weekend we were handed out Arduinos. To put it simply, it sucks. The code is horrible, and I would never write anything like it now. But this is not a failure, this is a testament to how much I have improved over this year in all three of the aspects of ACES. Looking at hardware, I just managed to send information through the air between two coils. A year ago, I did not even know what a coil was. For design, I had only stared in wonder at what others did with their CAD tools until this year. I remember coming in after class last year, asking how the simple on/off switch on a PCB we were handed worked, finding it incredible. Now, not only have I designed multiple PCBs, but I have also moved on to dip my toes into 3D printing. Similarly to my software, my 3D designs did not start off as masterpieces. My first print was the case for the Data Logger, and the screw holes were too small, the walls were too thick, and the case was almost too big for the Perma-Proto board I was using. On the other hand, my recent 5 V relay project uses an intricate stand that first perfectly with moving parts.

The point of this is not to say how much I know now, as I am well aware that there is still so much left to learn. My point is to express my gratitude to Hardware and to its teacher, Mr. D'Arcy, for always being interested in whatever I am working on and always encouraging my learning.

# ICS4U

# Project 3.1.1 CHUMP: Code, Clock, and Counter

## Theory

This project is this first of a series in which a 4-bit TTL (Transistor-Transistor-Logic) processor will be built from scratch. Named CHUMP (Cheap Homebrew Understandable Minimal Processor), this project aims to teach about low-level computer architecture. While CHUMP may be a relatively primitive computer compared to the ones that surround us in everyday life, the knowledge gained from CHUMP is highly applicable to more complex processors and CPUs.

## Purpose

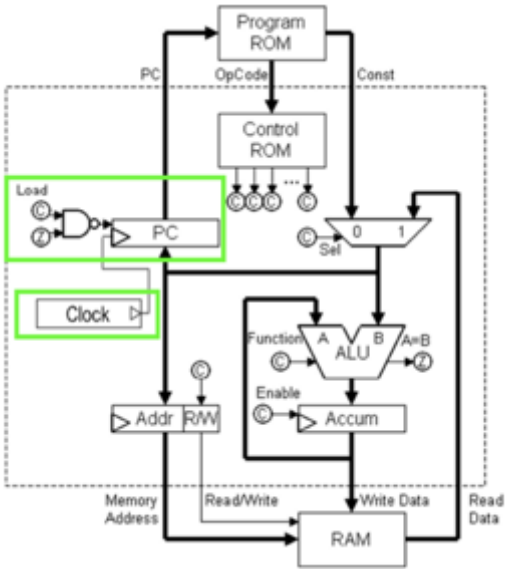
The purpose of this project is to set the foundations for the rest of CHUMP by implementing the clock and counter, as well as writing code to run on the finished processor. The clock and counter are fundamental to the working of the final build as together they synchronise all of the many components and keep track of what line of code the processor should currently be working on, keeping the entire build on track

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#tasks
Original Chump Paper: https://www.el-kalam.com/wp-content/uploads/2020/03/A-Simple-and-Affordable-TTL-Processor-for-the-Classroom.pdf

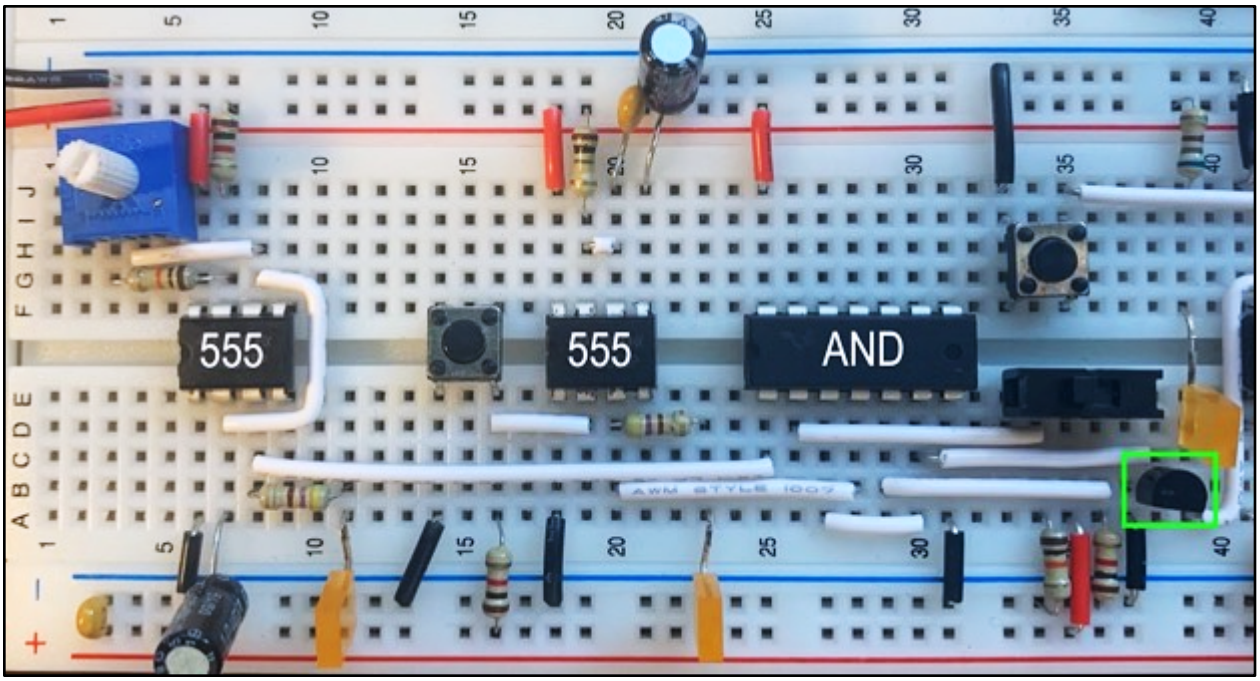
## Procedure

As described, this project begins CHUMP with the clock and program counter. In the image to the right, you can see their position and importance in the processor. The clock is especially important, as every triangle symbol on different components represents an input from the clock signal. These clock line connections ensure that all components run on time. The program counter takes the clock input and keeps track of the current line being executed in code, which it outputs as a 4-bit value to the program ROM to use. In addition, the program counter has external logic to implement jumping instructions, which take pre-set values and make them the program counter output. This logic decides when to use the external values or when to only focus on the clock signal.




## Clock

Before discussing the clock, it is important to know the design specifications imposed on it. The clock must be able to output an automatic square-wave with varying frequency, but with the flip of a switch change modes and be controlled by a button input. To accomplish this, two 555 timers from last year are implemented. One in astable mode, and one in monostable mode. To turn these two timers into a single output, an AND gate is used to essentially mask each signal, and the signal is selected by a slide switch. The output from the NAND gate is then amplified by a transistor to minimize the effect of fanout across the circuit.



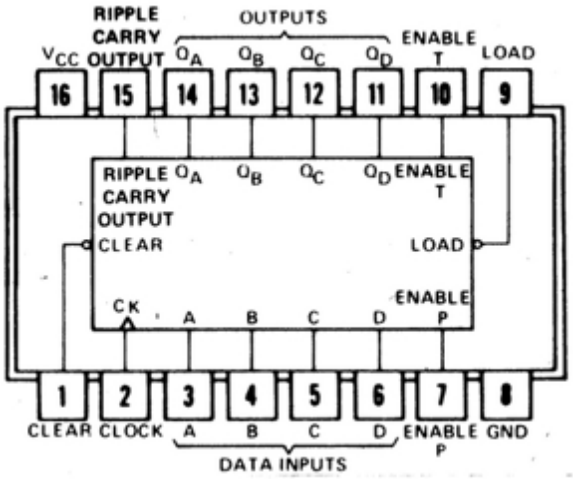

Above is a picture of the clock circuit built up on a breadboard. All the components have been previously explained in past projects, so there is no need to re-explain them. To the left is the first 555. This timer is in astable mode, meaning it outputs a constant square-wave on pin 3. The potentiometer in the top left effects the time constant in the RC circuit of the 555, meaning that when it turns the frequency of the output will change. The next 555 is in monostable mode, where it essentially acts as an over-engineered button de-bouncer for the manual stepping button. Both 555 outputs then go into the AND gate where one is masked with the slide switch's signal. This AND gate cleans up the signal while the slide switch is being switched due to the switch's open-before-closed nature.

One unique aspect of this version of the clock circuit is the 3904-transistor highlighted by the green rectangle, which amplifies the output of the AND gate to reduce the effect of fanout across the entire build. One thing to note about this setup is that the transistor will invert the signal of the AND gate, but this is without consequence as the output will still be an identical square-wave to the input, just with opposite phase.

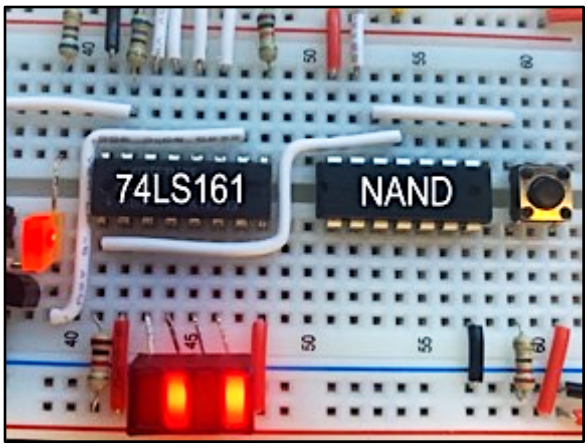
| **Parts Table** | |
|---|---|
| **Clock** | N/A |
| 555 Timer | 2 |
| 74LS08 AND Gate | 1 |
| 100 kΩ Potentiometer | 1 |
| 10 µF Polarized Capacitor | 2 |
| 0.1 µF Ceramic Capacitor | 1 |
| SPDT Slide Switch | 1 |
| Push Button (Normally Open) | 1 |
| 3904 NPN Transistor | 1 |
| Rectangle 5 mm LEDs | 3 |
| Assorted Resistors | ** |
| Assorted Wires | ** |

## Counter

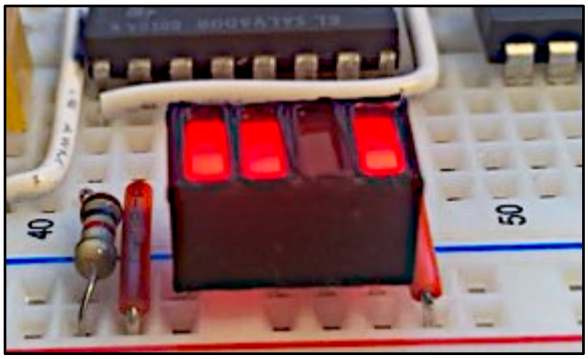
The counter circuit is built around the 74LS161, which is a 4-bit BCD synchronous counter. Similar to the 4516, the 74LS161 will increment its count by one on every rising edge, and will rollover when it hits its max value (the 74LS161 has a maximum value of 15 compared to the 9 of the 4516). In addition to this function, the counter can also load a pre-set value from the A, B, C, and D inputs if the load pin goes low. This is important, as by doing so the program counter can implement jump instructions, which are the basics of branching and complex programs. Finally, the counter also has a pin named Enable P (pin 7) This pin is normally high, meaning that counting is enabled. However, when low, counting stops. This is how the halt function is implemented.




To the right is the counter on the breadboard. Note that the chip itself is flipped, so the output pins $Q_A$-$Q_D$ are to the bottom. As seen, the clock's output can be seen going into pin 2. The four white wires on the input pins are all connected to slide switches on a different breadboard to control the LOAD value. The bottom left LOAD pin can be seen being controlled by a NAND gate. This NAND gate will eventually take two inputs (one from the ALU and one from Control EEPROM) to control when the set the LOAD pin low, which should be when both inputs to the NAND gate are high. Right now, the inputs are connected to a button.




One addition to the normal circuit is the 3D printed sheath for the rectangle LEDs. It is not too noticeable, but it does a nice job of making sure that all the LEDs are nicely aligned, which was a slightly annoying problem when they were not. Overall, they do not have a big effect on the circuit but they do look pretty cool compared to the bare, unaligned LEDs, and they almost make the output of the program counter look like a 4-bit bar graph.



To the right are the parts used for the program counter. Now is a good time to point out the discrepancy between this clock and counter compared to Ben Eater's, which works but is very overengineered. This design works perfectly, as well as combating fanout. Overall, it is worth the saved space and components.

| **Parts Table** | |
|---|---|
| **Program Counter** | N/A |
| 74LS161 BCD Counter | 1 |
| 74LS00 NAND Gate | 1 |
| Push Button (Normally Open) | 1 |
| SPDT Slide Switch | 2 |
| Rectangle 5 mm LEDs | 4 |
| Assorted Resistors | ** |

## Code (Chump Code)

Before looking at the CHUMP code for this project, we need to learn the language of CHUMPanese, which is essentially a custom assembly language for our CHUMP. Since data busses and addresses are only 4 bits wide, there can only be 16 possible instructions ($2^4 = 16$). Each instruction will have two flavours: one for constant operations, and one for memory operations. This means that any instruction can take the value of some address in memory as its input, or a constant. Constant operations have 0 as their LSB in the op-code, while memory operations have 1 as their LSB. The complete instruction set is as follows:

| OpCode (Machine) | Operand (Const/Mem) | Assembly | High-Level |
|---|---|---|---|
| 0000 | Const | LOAD | accum = const; pc++ |
| 0001 | Mem | | accum = mem[addr]; pc++ |
| 0010 | Const | ADD | accum += const; pc++ |
| 0011 | Mem | | accum += mem[addr]; pc++ |
| 0100 | Const | SUBTRACT | accum -= const; pc++ |
| 0101 | Mem | | accum -= mem[addr]; pc++ |
| 0110 | Const | STORETO | mem[const] = accum; pc++ |
| 0111 | Mem | | mem[addr] = accum; pc++ |
| 1000 | Const | READ | addr = const; pc++ |
| 1001 | Mem | | addr = mem[addr]; pc++ |
| 1010 | Const | IFNEG | accum < 0 ? pc = const : pc++ |
| 1011 | Mem | | accum < 0 ? pc = mem[addr] : pc++ |
| 1100 | Const | GOTO | pc = const; |
| 1101 | Mem | | pc = mem[addr]; |
| 1110 | Const | IFZERO | accum==0?pc=const : pc++ |
| 1111 | Mem | | accum==0?pc=mem[addr] : pc++ |

One thing to note is the lightly shaded `IFNEG` instruction. This is the custom instruction that is planned to be implemented into the CHUMP build. It will jump if the output of the accumulator is negative. Without getting too far ahead, this can be checked using the $C_{n+4}$ output pin on the accumulator during subtraction operations (called a borrow when subtracting instead of a carry), so it should be relatively easy to implement.

There are some other instructions that are important to understand as well. `IFZERO` will jump (set the program counter) to a specific line if the *Z* flag is set high. This is a branching instruction, along with `GOTO`, meaning that these instructions control the path that the processor takes through the program. `IFZERO` is like a low-level `if()` statement, used for conditional branching, while `GOTO` will unconditionally jump to a specified line.

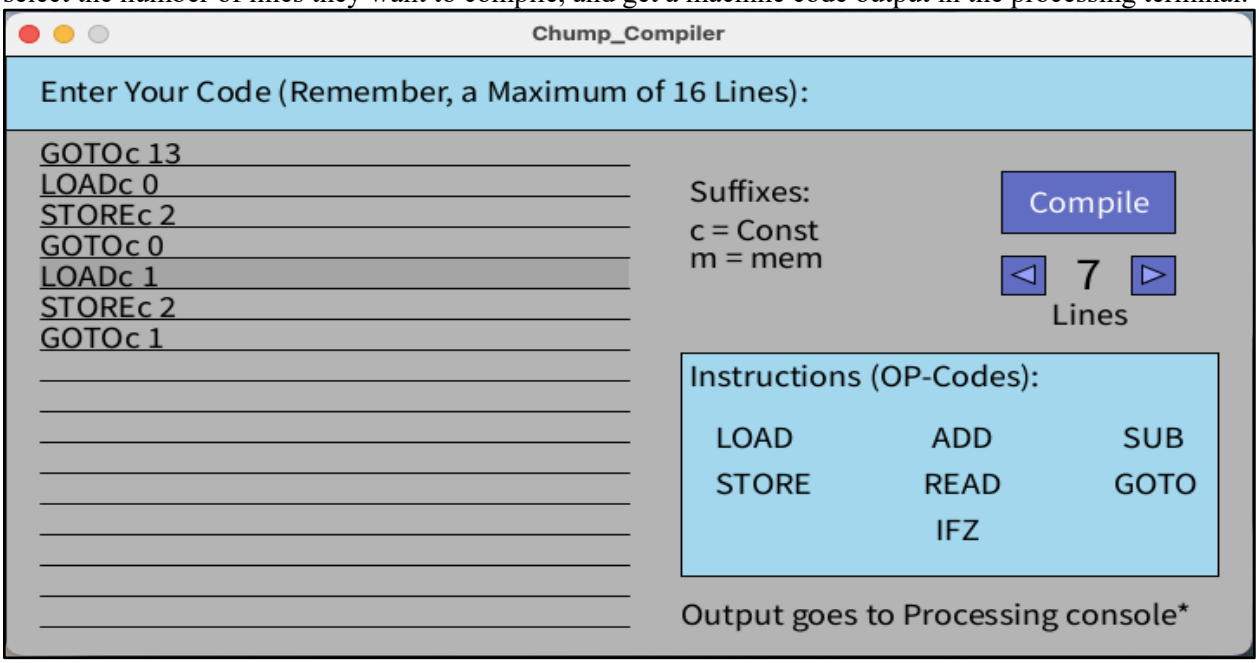
Next, the `READ` operation is important yet slightly confusing. `READ` sets the address register to either a constant or to the value of some location in memory, that address can then be loaded into the accumulator in the next instruction. This means `LOAD` (and some other instructions) should be preceded by a `READ`, but only if planning to load the value of some address in memory. To load a constant, this is unnecessary. As for the rest of the functions, they should be relatively straightforward in their workings. `ADD` and `SUBTRACT` respectively add or subtract their operand from the accumulator, and `STORETO` stores the accumulator's value to memory. Now with that, we can look at the code for this project.

| Line | Machine Code | Hex | Assembly Operation & Operand | High Level |
|---|---|---|---|---|
| 0000 | 00000101 | 0x05 | LOAD 5 | accum <- 5, pc++ |
| 0001 | 01100000 | 0x60 | STORETO 0 | mem[0] <- accum, pc++ |
| 0010 | 00000111 | 0x07 | LOAD 7 | accum <- 7, pc++ |
| 0011 | 01100001 | 0x61 | STORETO 1 | mem[1] <- accum, pc++ |
| 0100 | 10000000 | 0x80 | READ 0 | addr <- 0, pc++ |
| 0101 | 00010000 | 0x10 | LOAD it | accum <- mem[addr], pc++ |
| 0110 | 10000001 | 0x81 | READ 1 | addr <- 1, pc++ |
| 0111 | 01010001 | 0x51 | SUBTRACT it | accum <- accum - mem[addr], pc++ |
| 1000 | 10101010 | 0xAA | IFNEG 10 | accum < 0 ? pc <- 10 : pc++ |
| 1001 | 11001101 | 0xCD | GOTO 13 | pc <- 13 |
| 1010 | 00000000 | 0x00 | LOAD 0 | accum <- 0, pc++ |
| 1011 | 01100010 | 0x62 | STORETO 2 | mem[2] <- accum, pc++ |
| 1100 | 11000000 | 0xC0 | GOTO 0 | pc <- 0 |
| 1101 | 00000001 | 0x01 | LOAD 1 | accum <- 1, pc++ |
| 1110 | 01100010 | 0x62 | STORETO 2 | mem[2] <- accum, pc++ |
| 1111 | 11000001 | 0xC1 | GOTO 0 | pc <- 0 |

What this code does is compare two inputs (on line 0000 and 0010) and determine whether input *a* is more than input *b*. In this case, the inputs are *a = 5* and *b = 7*, so the output (memory address 2) will be set to 0 as *a* is not greater than *b*. If *a* had been greater, then the output would have been 1. To accomplish this, the program takes the two variables, stores them, and then subtracts *b* from *a*. If this result is negative, the output memory address is loaded with 0. Otherwise (when *a > b*) the output address is loaded with 1. To check if the result is negative, the custom `IFNEG` function is used.

## CHUMP IDE

The process of writing machine code for CHUMP is quite tedious. The CHUMP compiler (or assembler) aims to change that. Below is an image of the program, written in Java. The program is launched by a Processing sketch which opens this GUI window where the user can type their assembly-level CHUMP, select the number of lines they want to compile, and get a machine code output in the processing terminal.



To the right is a picture of the output stream in the Processing terminal. Here you can see the error catching which stops the program from crashing if there is a typo in the code when it tries to compile. Beneath that is a stream of machine code from a successful compiling which can be used for CHUMP.

While programing in pure machine code for a maximum of 16 lines is manageable without an IDE, the process of creating a GUI application such as this one taught a lot about programing.

```
Error Caught! Stop Making Errors Please!
(This usually means that you are either
trying to compile unwanted lines or you
have a typo)
Error was found on line 8.
11001101
00000000
01100010
11000000
00000001
01100010
11000001
```

## Code

Shortened Compiler Code (Full working code is on the project's GitHub):

```
//Project : CHUMP Compiler
//Date    : Sep 9, 2023
//Author  : Liam McCartney
//Status  : Working on GitHub

String userInput[] = {"", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
""}; // Store the user input

String LOADc = "0000";
String LOADm = "0001";

String ADDc = "0010";
String ADDm = "0011";

String SUBc = "0100";
String SUBm = "0101";

String STOREc = "0110";
String STOREm = "0111";

String READc = "1000";
String READm = "1001";

String GOTOc = "1100";
String GOTOm = "1101";

String IFZc = "1110";
String IFZm = "1111";
//Chump language

boolean button1Pressed;
boolean button2Pressed;
boolean button3Pressed;
//Variables

int lines = 0;

void setup() {
  size(720, 390);
  background(255, 255, 255);
  stroke(150, 216, 240);
  rect(0, 0, 720, 64);

  fill(0);
  textSize(20);
```

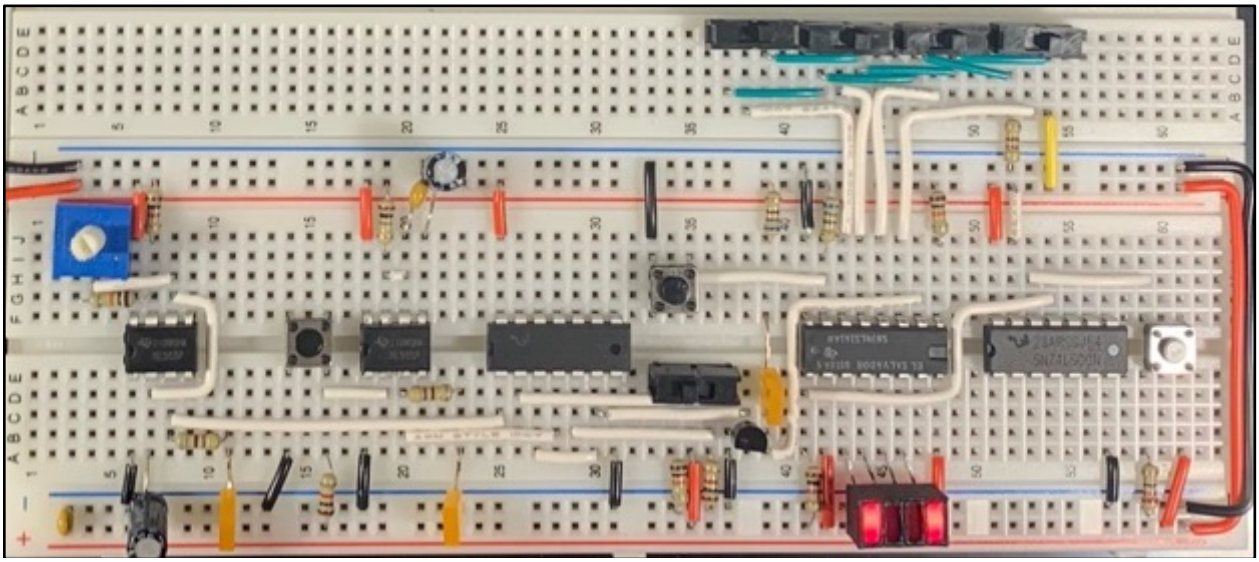
```
  text("Enter Your Code (Remember, a Maximum of 16 Lines): ", 20, 20);
  //Set the background and look
}

void draw() {
  textAlign(LEFT, BASELINE);
  background(220);
  fill(180, 180, 180);
  rect(0, 48, 720, 342);

  fill(150, 216, 240);
  rect(0, 0, 720, 48);
  rect(390, 193, 310, 145);

  fill(165, 165, 165);
  noStroke();
  rect(20, 52 + 20 * current, 340, 20);
  stroke(0);
  fill(0);

  text("Suffixes:", 395, 95);
  text("c = Const", 395, 120);
  text("m = mem", 395, 138);

  text("Enter Your Code (Remember, a Maximum of 16 Lines): ", 20, 30);

  text("Instructions (OP-Codes):", 395, 215);
  //Set up base GUI

  if (lines != 1) {
    text("Lines", 604, 175);
  } else {
    text("Line", 609, 175);
  }
  //It bugs me when people are too lazy to fix stuff with 1 and then a plural

  text("Output goes to Processing console*", 390, 370);

  text("LOAD", 410, 255);
  text("ADD", 535, 255);
  text("SUB", 645, 255);
  text("STORE", 410, 285);
  text("READ", 530, 285);
  text("GOTO", 640, 285);
  text("IFZ", 537, 315);
  //OpCode TOC

  text(userInput[0], 20, 55 + offset);
  text(userInput[1], 20, 75 + offset);
  text(userInput[2], 20, 95 + offset);
  //Etc, etc. This happens 16 times, full code on Github
  //Write the lines

  textSize(30);
  if (lines < 10) {
    text(lines, 618, 152);
  } else {
    text(lines, 611, 152);
  }
  textSize(20);
  //Line number selection

  fill(100, 100, 200);
```

```
  rect(575, 75, 100, 40);
  //Button

  if (mouseX > 575 && mouseX < 675 && mouseY > 75 && mouseY < 115) {
    fill(150, 150, 255); //Highlight the button when the mouse is over it
  } else {
    fill(100, 100, 200); //Use the default button color when it is not
  }
  rect(575, 75, 100, 40);
  rect(575, 130, 25, 25);
  //Do this again for every button… (I removed it for the DER copy, go to Github)

  //Display button text
  fill(255); // Text color
  textAlign(CENTER, CENTER);
  text("Compile", 626, 95);

  if (button1Pressed) {
    //Compile button
    compile();
    button1Pressed = false;
  //x3, once per button
}

void keyPressed() {
  if (key == BACKSPACE && userInput[current].length() > 0) {
    //If the user presses the BACKSPACE key and there is something to delete
    userInput[current] = userInput[current].substring(0, userInput[current].length()
- 1);
  } else if (key == BACKSPACE && userInput[current].length() == 0 && current != 0) {
    current--;
  } else if (keyCode == UP && current != 0) {
    current--;
  } else if (keyCode == DOWN && current != 15) {
    current++;
  }
}

void keyTyped() {
  if (key == &apos;\n&apos;) {
    current++;
  } else if (key >= ` `; &apos; && key <= `z`;) {
    //If the typed character is a printable character, add it to userInput
    userInput[current] += key;
  }
}

void mousePressed() {
  //Check if the mouse is over the button when it's pressed
  if (mouseX > 575 && mouseX < 675 && mouseY > 75 && mouseY < 115) {
    button1Pressed = true;
  } else if (mouseX > 575 && mouseX < 600 && mouseY > 130 && mouseY < 155) {
    button2Pressed = true;
  } else if (mouseX > 650 && mouseX < 675 && mouseY > 130 && mouseY < 155) {
    button3Pressed = true;
  } else if (mouseX > 20 && mouseX < 360 && mouseY > 55 && mouseY < 371) {

    int scaledMouse = mouseY - 55;
    int page = scaledMouse / 20;
    current = page;
  }
}
```

```
void compile() {
  for (int i = 0; i < lines; i++) {
    String str = userInput[i];
    //isolate the line we are working on
    String[] subStr = str.split(" ");
    //Very useful function to split the string right where we need it

    if (subStr[0].contains("LOADc")) {
      print(LOADc);
    } else if (subStr[0].contains("LOADm")) {
      print(LOADm);
    } else if (subStr[0].contains("ADDc")) {
      print(ADDc);
    } else if (subStr[0].contains("ADDm")) {
      print(ADDm);
    } else if (subStr[0].contains("SUBc")) {
      print(SUBc);
    } else if (subStr[0].contains("SUBm")) {
      print(SUBm);
    } else if (subStr[0].contains("STOREc")) {
      print(STOREc);
    } else if (subStr[0].contains("STOREm")) {
      print(STOREm);
    } else if (subStr[0].contains("READc")) {
      print(READc);
    } else if (subStr[0].contains("READm")) {
      print(READm);
    } else if (subStr[0].contains("GOTOc")) {
      print(GOTOc);
    } else if (subStr[0].contains("GOTOm")) {
      print(GOTOm);
    } else if (subStr[0].contains("IFZc")) {
      print(IFZc);
    } else if (subStr[0].contains("IFZm")) {
      print(IFZm);
    }
    //There has to be a better way to do this....
    //Checks which function it is and prints the value

    try {
      //Error catching! Yay!
      int containedInt = Integer.parseInt(subStr[1]);
      //Java turns it into an integer for me!

      String binaryNibble = String.format("%4s",
Integer.toBinaryString(containedInt)).replace(&apos; &apos;, &apos;0&apos;);
      //Java has a built in function for everything

      println(binaryNibble);
    }
    catch(ArrayIndexOutOfBoundsException e) {
      //Oh O. You messed up :(
      println("Error Caught! Stop Making Errors Please!");
      println("(This usually means that you are either trying to compile unwanted
lines or you have a typo)");
      print("Error was found on line ");
      print(i + 1);
      println(".");
    }
  }
}
```

## Media

Project Video: https://youtu.be/gs3cXJvP98k
Full Processing Sketch (Project Git): https://github.com/Liam-McCartney/Hardware/tree/main/CHUMP



Clock and Counter

## Reflection

I cannot deny that the idea of CHUMP has intimidated me for some time. While rationally I always knew that if others before me could do it then so could I, it is just such a big step forward in complexity compared to anything we have done before. However, this project went great. I understood everything I was doing, I learned a lot, and it was fun to get back into all things Hardware. After this beginning, I now have a much more optimistic view of the remainder of CHUMP.

The other thing that I wanted to touch on was my CHUMP IDE, which I am quite proud of. It was my first ever GUI application, which is another thing that has intimidated me in the past. I am not sure what, but something on the very first day of Hardware spurred me to finally break the back of GUI applications, and I am happy with the final product. In making the CHUMP IDE, I learned an incredible amount about user-friendly GUI applications, compared to a mess of output LEDs that normally signify CHUMP. While it may not be perfect, that is to be expected with my very first attempt, and I look forward to working more on this new skill.

# Project 3.1.2 CHUMP: Program and Control EEPROM

## Purpose

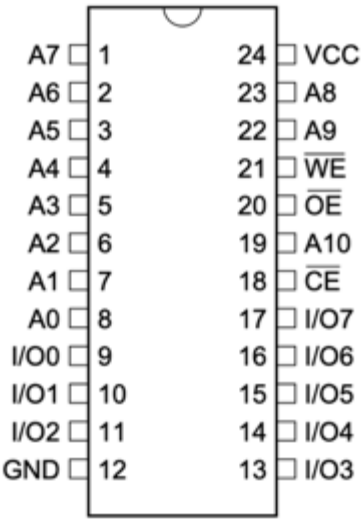
The purpose of Project 3.1.2 is to implement the two EEPROM ICs into the CHUMP build. One will be used as memory to store the program that the processor will run while the other will demonstrate a more general concept of EEPROM's ability to replace combinational logic by acting as a programmable logic chip.

## References

Project Description:
http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#PROGRAMEEPROM
AT28C16 Datasheet: http://darcy.rsgc.on.ca/ACES/Datasheets/AT28C16.pdf

## Procedure

As mentioned above, this project revolves around the implementation of EEPROMs into CHUMP. However, the EEPROM ICs cannot simply be wired up within CHUMP and expected to perform correctly. They must first be programmed using an EEPROM programmer, which in this case is simply an Arduino Nano.

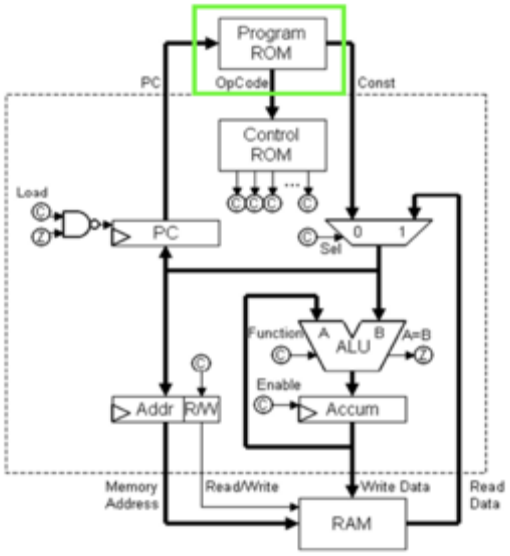



While the programmer is not the subject of this project, it is useful to know how to program the EEPROM by understanding the EEPROM IC, the AT28C16, and its pinout. To begin with, the AT28C16 is a two-kilobyte parallel EEPROM chip. It has $2^{11}$memory addresses, each eight bits wide, which is enough to store 128 ($2^7$) CHUMP programmes as a single program is $2^4$ bytes long. The function of this IC is quite simple when conditioned correctly. By wiring $\overline{OE}$ and $\overline{CE}$ low and $\overline{WE}$ high, the EEPROM is put in read mode. In this mode, the contents of the address in memory specified by inputs A0 to A10 is outputted on the eight I/O pins. The fact that the address is 11 bits wide allows for the implementation of manual user paging between different programmes in memory by changing the unused bits in the address line with slide switches. To write to the chip, simply reverse the previous pin configuration and the input presented on the I/O pins will be mirrored to the address present on the address bus of the EEPROM. It is possible to program the chip by hand, but instead an Arduino Nano is used to handle all the pin munipulation.

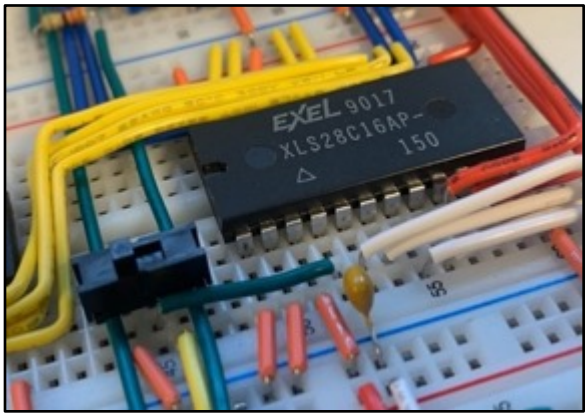
One of the less obvious properties of EEPROM is its ability to act as any logic gate. When thinking about logic gates or even complex combinational logic, the only thing that matters is the output for a given input. It does not matter what happens under the hood. EEPROMs can have this same functionality. By treating the address to read from as the input into the logic sequence, the value stored at that location can be thought of as the output for that given input address. Thus, by reading from that address, the value the EEPROM spits out will the desired output. While this works completely differently from combinational logic, it is much easier to implement and the only drawbacks are speed and cost. With this understanding of EEPROM complete, it is time to discuss this project in specific.

## Program EEPROM

The first implementation of EEPROM into this project is the program EEPROM, shown to the right in CHUMP's architecture diagram. In the diagram it is referred to as program ROM, as CHUMP will never edit its contents. The purpose of the program EEPROM is to store the program that the processor will run. As shown, the output of the program counter register is the input to the program EEPROM. The program EEPROM takes the value of the program counter and spits out the contents of that location in memory, which is the line of machine code meant to be executed. The high nibble (which is the op-code) is sent to the control EEPROM, and the low nibble (which is either a constant or address in memory) is sent to the multiplexer

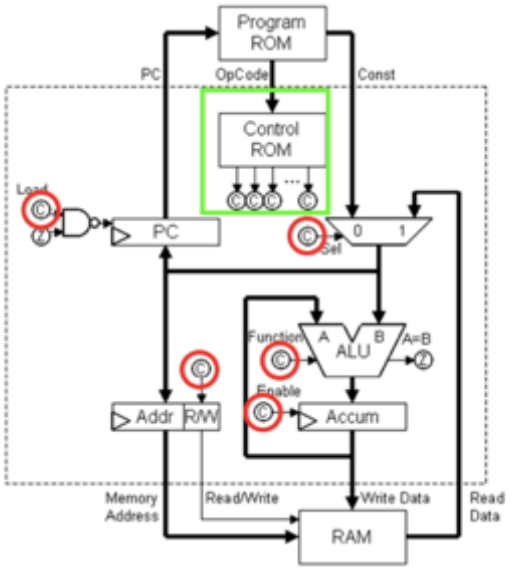



One thing to note is the width of the address bus into the program EEPROM. CHUMP is built around a 4-bit architecture, so all data and address lines are 4-bits wide. However, the AT28C16 has 11 address bits. This leaves 7 bits that can still be manipulated through external means, such as slide switches and buttons. For example, by taking the fifth bit and connecting it to a slide switch (shown on the right), the user can switch between programs by altering the fifth bit of the address. This allows the user to store multiple programs at once. In addition, by manipulating the ALU and with some extra logic, this makes it theoretically possible to extend CHUMP's 16-line limit on programs by allowing the processor to page itself, although this would be a whole other project in itself.




## Control EEPROM

While program EEPROM is relatively straightforward use of memory, control EEPROM is not. Control EEPROM replaces the Control Unit found in modern CPU architectures. The job of the control EEPROM is to make sure the processor runs smoothly and produces meaningful results instead of a jumble of random zeros and ones. The control EEPROM conditions the rest of the components within CHUMP to make sure they are all doing the correct operation. This is seen on the right through the C outputs of the control EEPROM, which are wired to all the highlighted pins of other components. By doing so, the control EEPROM coordinates the actions of CHUMP. However, to accomplish the control EEPROM needs correct control codes to output in the first place.



Deciding on control codes is not a trivial task. It is essential to remember that the control EEPROM can only output a maximum of eight control bits, so control codes must be highly optimised. A table of all the possible instructions and the required outputs is a good place to start.

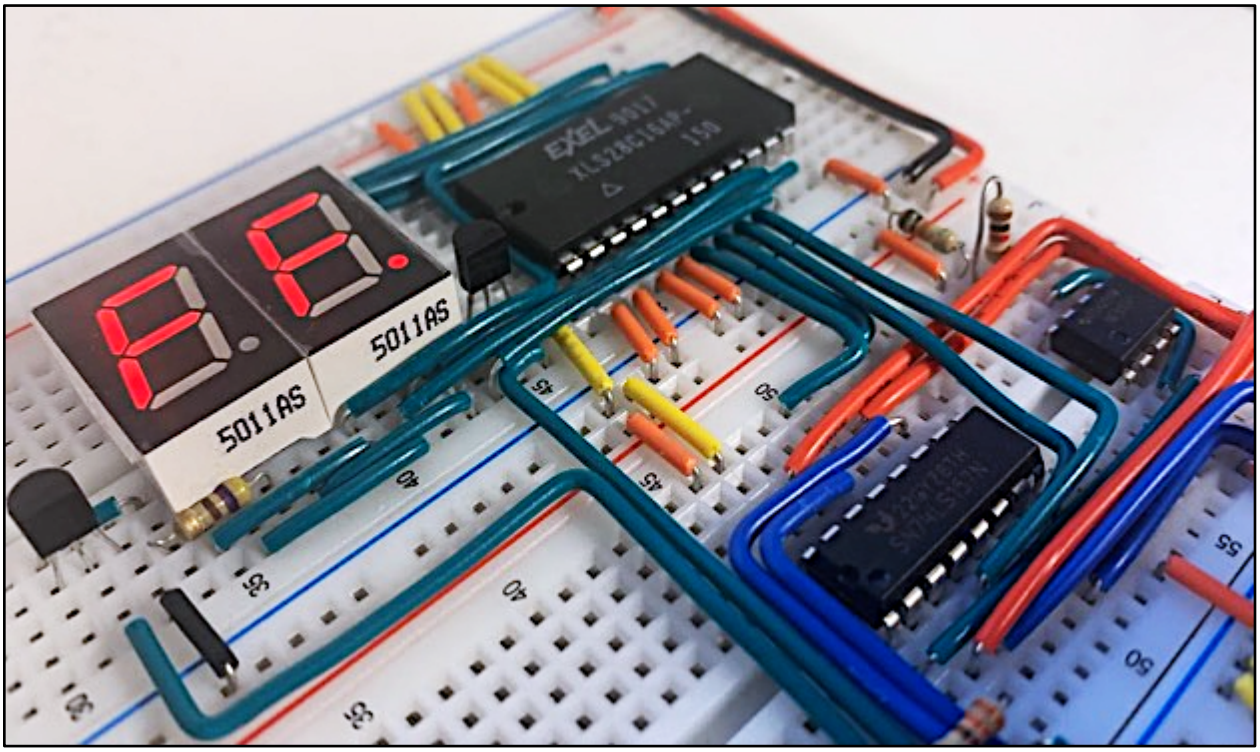
| Instruction | Op3 | Op2 | Op1 | Op0 | Sel | ALU | S3 | S2 | S1 | S0 | M | $C_n$ | Acc | R/W | PC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LOAD c | 0 | 0 | 0 | 0 | 0 | B | 1 | 0 | 1 | 0 | 1 | X | 0 | 1 | 0 |
| LOAD it | 0 | 0 | 0 | 1 | 1 | B | 1 | 0 | 1 | 0 | 1 | X | 0 | 1 | 0 |
| ADD c | 0 | 0 | 1 | 0 | 0 | A plus B | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| ADD it | 0 | 0 | 1 | 1 | 1 | A plus B | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| SUB c | 0 | 1 | 0 | 0 | 0 | A – B | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| SUB it | 0 | 1 | 0 | 1 | 1 | A – B | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| STORE c | 0 | 1 | 1 | 0 | 0 | A | 1 | 1 | 1 | 1 | 1 | X | 0 | 0 | 0 |
| STORE it | 0 | 1 | 1 | 1 | 1 | A | 1 | 1 | 1 | 1 | 1 | X | 0 | 0 | 0 |
| READ c | 1 | 0 | 0 | 0 | 0 | A | 1 | 1 | 1 | 1 | 1 | X | 1 | 1 | 0 |
| READ it | 1 | 0 | 0 | 1 | 1 | A | 1 | 1 | 1 | 1 | 1 | X | 1 | 1 | 0 |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| GOTO c | 1 | 1 | 0 | 0 | 0 | *Logic 1* | 1 | 1 | 0 | 0 | 1 | X | 1 | 1 | 1 |
| GOTO it | 1 | 1 | 0 | 1 | 1 | *Logic 1* | 1 | 1 | 0 | 0 | 1 | X | 1 | 1 | 1 |
| IFZERO c | 1 | 1 | 1 | 0 | 0 | $\overline{A}$ | 0 | 0 | 0 | 0 | 1 | X | 1 | 1 | 1 |
| IFZERO it | 1 | 1 | 1 | 1 | 1 | $\overline{A}$ | 0 | 0 | 0 | 0 | 1 | X | 1 | 1 | 1 |

With this table some key patterns become recognizable. For example, the select input to the multiplexer is equivalent to the least significant bit in the op-code, so Sel does not require a control line among the final eight. In addition, PC can be calculated on the fly by putting the two most significant bits of the op-code through an AND gate. This leaves eight remaining control bits, which fits the EEPROM's output capabilities. However, not all eight are necessary. One pattern that is difficult to see is that S0 and $C_n$ are equivalent for the only four instructions where the value of $C_n$ is important. Thus, $C_n$ can be eliminated as well, freeing up an extra control line for a custom instruction. While this will be used in the final CHUMP build, for now the $C_n$control line will be kept for consistency. Below is a chart of the final control codes.

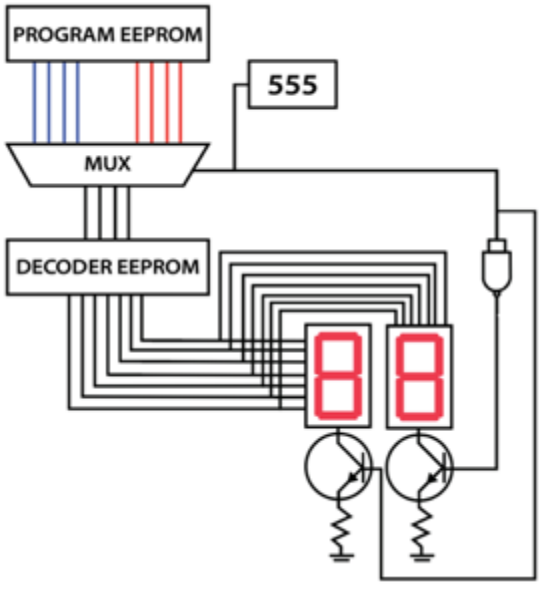
| Instruction | S3 | S2 | S1 | S0 | M | $C_n$ | Acc | R/W |
|---|---|---|---|---|---|---|---|---|
| LOAD c | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| LOAD it | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| ADD c | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| ADD it | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| SUB c | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| SUB it | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| STORE c | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| STORE it | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| READ c | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| READ it | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| | | | | | | | | |
| | | | | | | | | |
| GOTO c | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| GOTO it | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| IFZERO c | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| IFZERO it | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

## Program Display

This project sees the implementation of a third EEPROM in addition to program and control. This third EEPROM is used to decode four-bit numbers into outputs for a seven-segment display. These displays are used to show the current instruction being executed. Below is an image of the display.



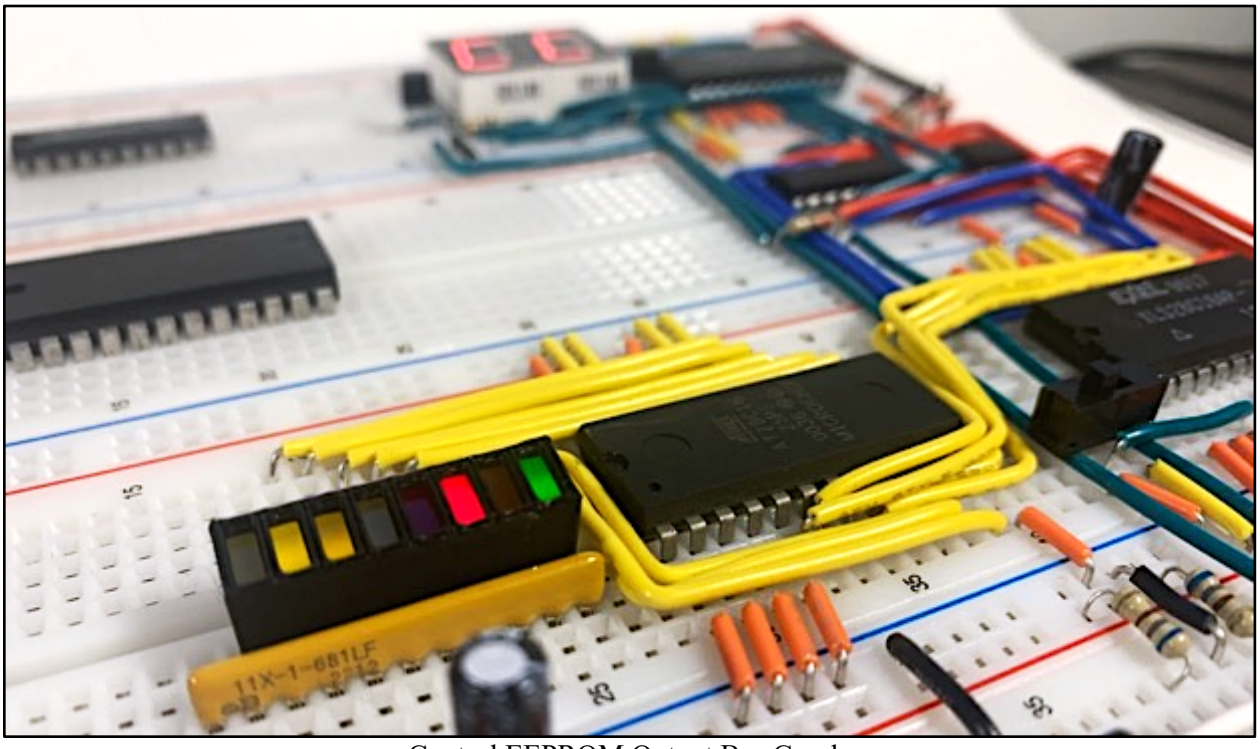

Normally it would take two EEPROMs to drive two seven-segment displays, but since that takes up quite a bit of breadboard real-estate as well as costing quite a bit, an alternative method was implemented. Instead of having two EEPROMs constantly powering two seven-segment displays, both the EEPROMs and seven-segment displays are multiplexed so that the single EEPROM outputs both display values. In addition to this, each common cathode of the seven-segment displays is also multiplexed such that they are of opposite phase, meaning only one display is on at any given time. The multiplexer is run by a 555 timer with a frequency of roughly 480 Hz, which is well within the switching range of all the components used. To the right is a diagram of exactly how this works. This approach makes use of a multiplexer, which will be explained in its own report.



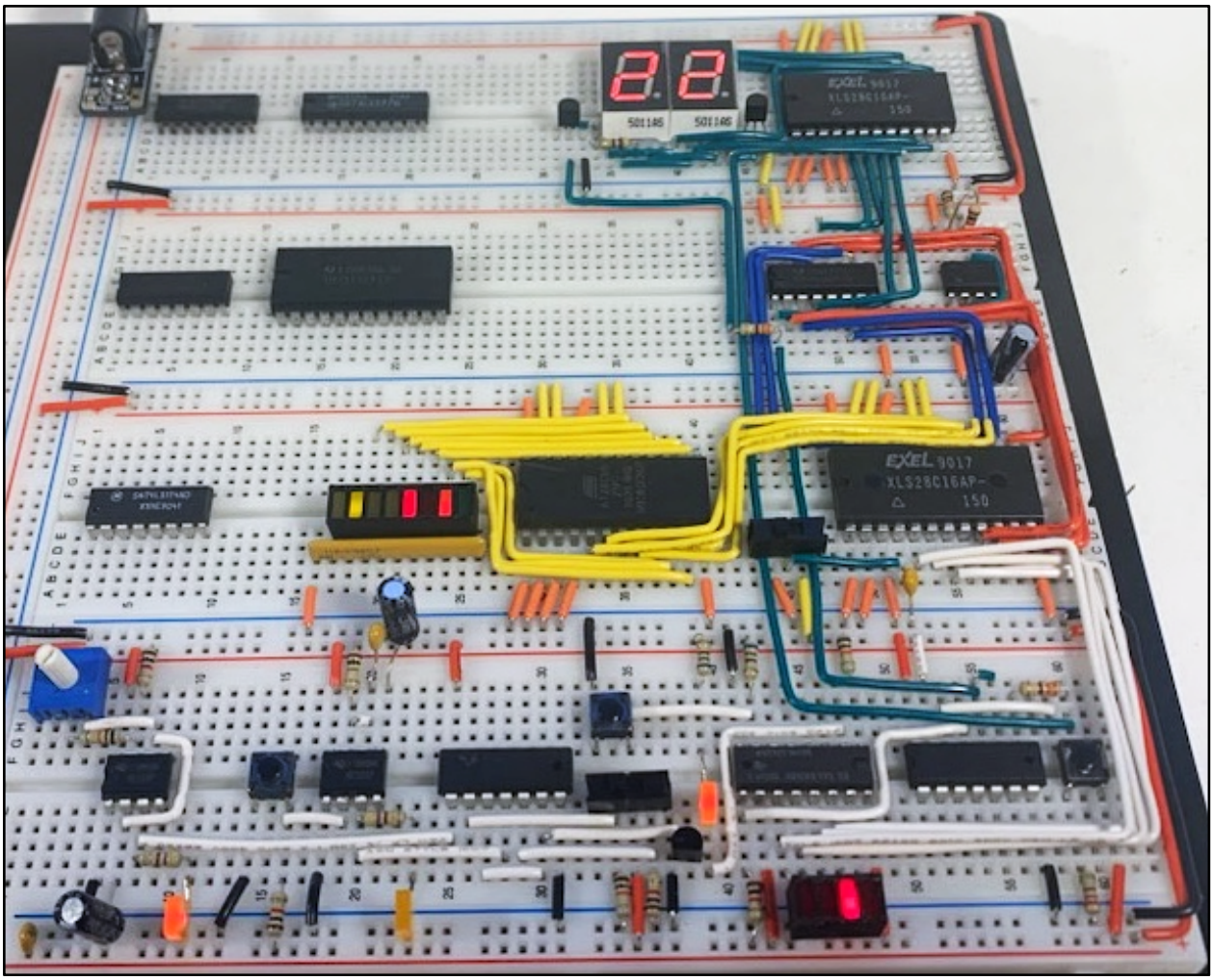
| Parts Table (Program EEPROM, Control EEPROM, & Display) | |
|---|---|
| AT28C16 Parallel EEPROM | 3 |
| 74LS157 Two-Channel Multiplexer | 1 |
| 555 Timer | 1 |
| 3904 NPN BJT Transistor | 2 |
| Common Cathode Seven-Segment Display | 2 |
| SPDT Slide Switch | 1 |
| 3D Printed 8-Bit Bar-Graph Sheath | 1 |
| 5 mm Rectangle LEDs | 8 |
| Bussed Resistor Network | 1 |
| 470 Ω Fixed Resistor | 1 |
| 330 Ω Fixed Resistor | 2 |
| 1 kΩ Fixed Resistor | 2 |
| 1 µF Polarized Capacitor | 1 |
| 0.1 µF Ceramic Capacitor | 1 |
| Assorted Wires | ** |

## Media

Project Video: https://youtu.be/ntZKFu_rGVM



Control EEPROM Output Bar-Graph




Current CHUMP Build

## Reflection

This project was quite a fun one for me. While the project itself of program and control EEPROM went relatively quickly, the task of multiplexing seven-segment displays completely in hardware was a good challenge, and it was amazing to see my original late-night sketch of the circuit come to life on the breadboard and work as intended. It is eye opening to do these hardware tasks that are so easy with software; it really shows the power of microcontrollers which I normally do not fully appreciate.

# Project 3.1.3 CHUMP: Arithmetic and Logic Unit

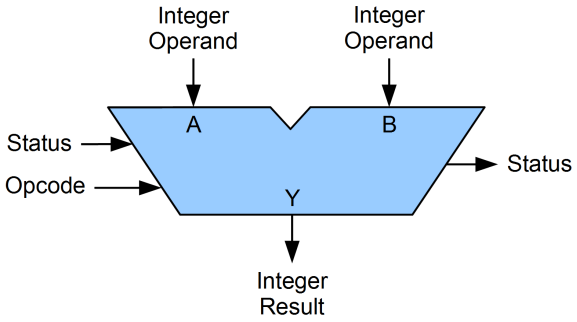
## Purpose

The purpose of this project is to explore the functioning of the 74LS181 arithmetic and logic unit (ALU) as well as to understand the function and purpose of ALUs in general within CPU architecture. While this project is separate from the rest of the CHUMP build, it will also begin to illuminate how the final build will handle branching instructions through the important idea of ALU flags.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#ALU
Ken Sherriff 74LS181 Blog: https://www.righto.com/2017/03/inside-vintage-74181-alu-chip-how-it.html

## Procedure

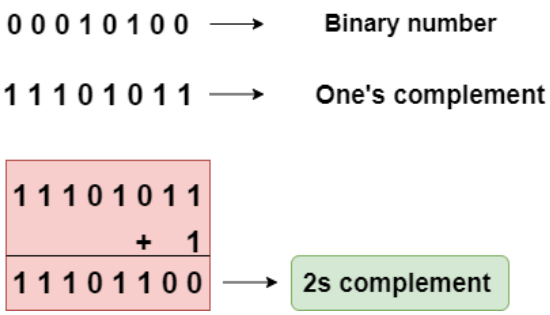



While this project focuses on a very specific ALU, the 74LS181, it is worthwhile to discuss the broader concept of ALUs in general. An ALU is a component in a computer's CPU that does as the name suggests: completes arithmetic and logical operations. The ALU is the place where the numbers are crunched within a computer. All ALUs have some common features, as shown on the right. To begin with, there will always be an A and B input. To manipulate those two inputs, the ALU takes in a control code (labelled op-code) which tells it which of its many operations to execute. The result of the operation is sent to the Y output. In addition, most ALUs will have flags (labelled status) which are bits that will be set high under certain conditions. The most common flags are Zero flags, Negative flags, Carry flags, and Overflow flags. If the output of the ALU matches any of those descriptions, the corresponding flag will be set high. It is important to note that different ALUs have different flags, and some have more than others.

### Building an ALU

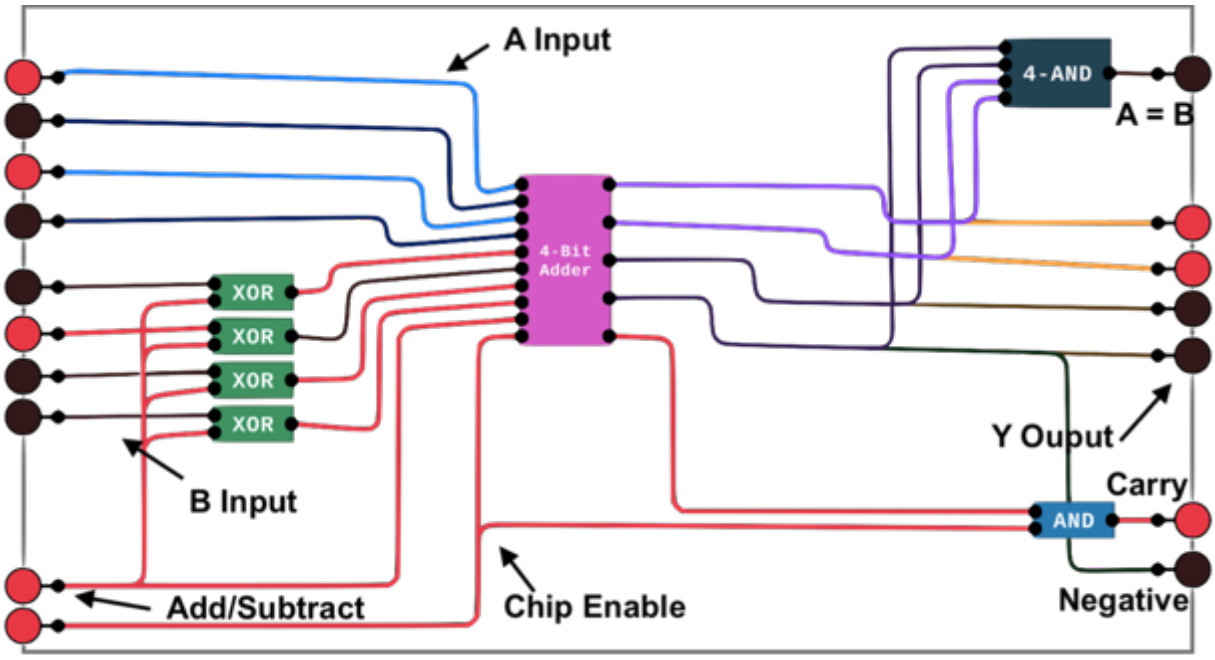
This description of ALUs is about as much knowledge as is necessary to complete this task for CHUMP, but it is still very abstracted from the bare bones of logic gates. ALUs are very complex and interesting devices, and it would be a missed opportunity to not explore their lowest-level inner workings. One way to do this exploration is by designing an ALU step by step.

To start designing an ALU it is important to set some parameters: firstly, this will be a 4-bit ALU for simplicity. In addition, the op-code will be 2-bits, once again to simplify the design. With these four possible op-codes, the ALU will be able to add, subtract, bitwise AND, and bitwise OR inputs A and B. Finally, there will be four flags: a carry flag, zero flag, negative flag, and a flag to mimic the misleading A = B flag on the 74LS181 which is in reality high when all output bits are high. Next, it is important to know that the general functioning of an ALU can be broken into two domains: logical and arithmetic operations. Within this ALU, both will be handled separately by an arithmetic unit (AU) and a logic unit (LU).



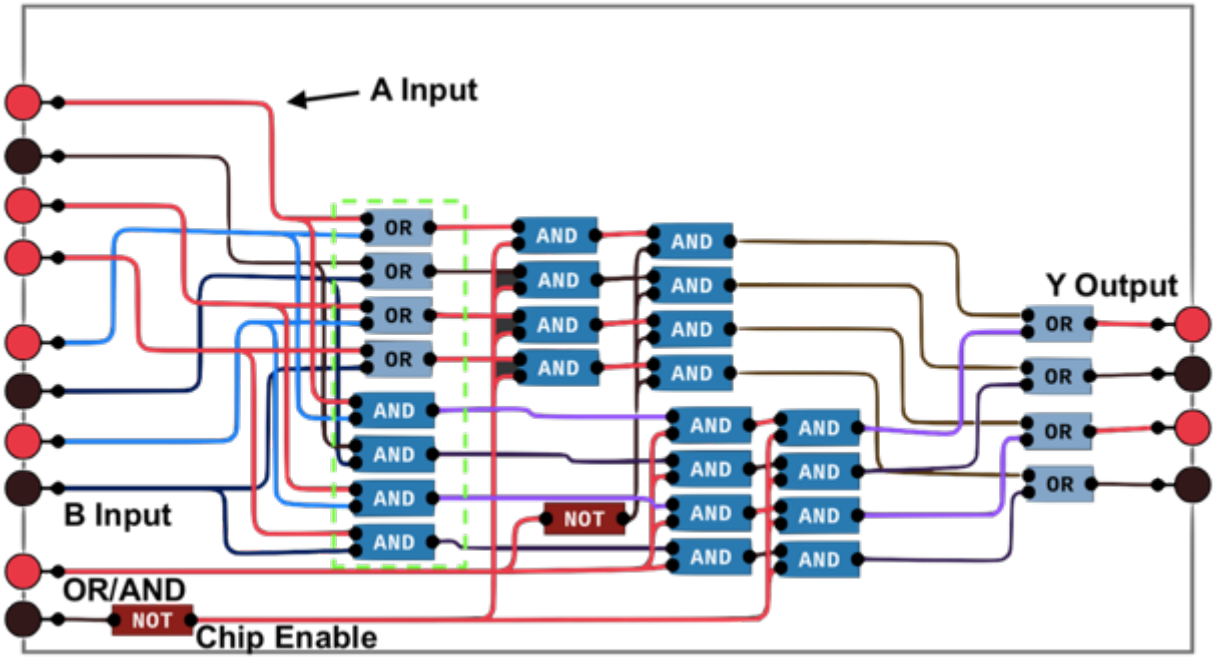

With that, it is time to begin designing. To build an AU capable of addition and subtraction is the first task. One nuance with this description is that subtraction is the same as the addition of a negative number to a positive number. However, up until now this report has not covered negative numbers represented in binary. To do this requires a system named two's complement. The basic idea of two's complement stems from the requirement of two numbers with equal magnitudes but opposite signs to sum to 0. Thus, this binary representation must uphold that mathematical truth. This means that the negative version of a binary number will be its compliment plus 1, such that all the bits will cancel out in addition to sum to 0. An alternate way of thinking about this is that the most significant bit in the nibble will represent -8 instead of 8.

Luckily, normal adder circuit work perfectly fine with this new two's complement system, and adder circuits have already been covered in this report. With only a slight alteration to a 4-bit adder, an AU capable of addition and subtraction is easily achievable. Below is an image of such an AU.

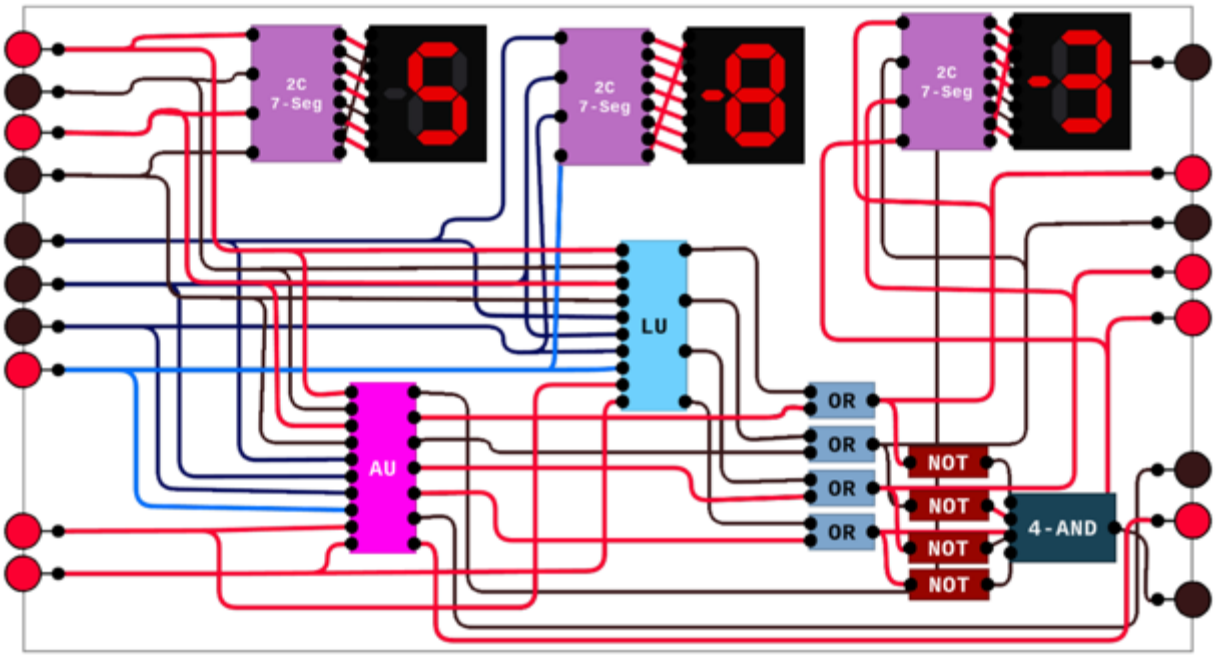



Notice the add/subtract line. When low, the adder will add the two inputs. However, when high, the carry input is turned high and the B inputs go through a series of XOR gates, getting inverted. This is the implementation of two's complement, where the B input is inverted and 1 is added through the carry input into the adder.

Next is the LU for the ALU. The LU is easier to implement as it simply means running the inputs through some logic gates. As mentioned, the LU will be able to perform bitwise ANDs and bitwise OR operations. Below is an image of the LU used in this custom ALU. While it may look quite complicated, the AND and OR logic occurs within the green rectangle, and the rest of the logic chooses between the AND/OR outputs depending on the input on the AND/OR control line and the chip enable pin. Note that chip enable is active low so that the LU is active when the AU is inactive, and vice versa.

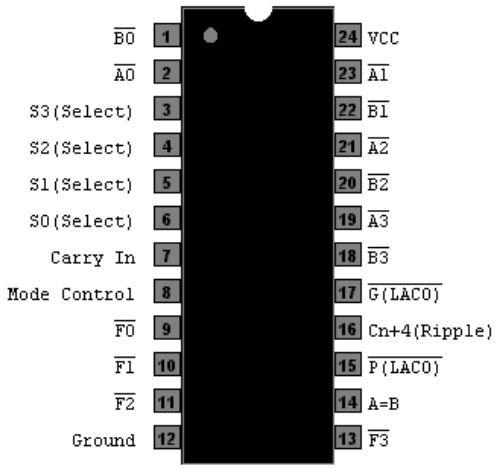



Below is a picture of the final ALU performing the operation 5 – 8. In addition to the Y output, the ALU also includes four flags as mentioned, with the negative flag currently high. While this is a much simpler ALU than the 74LS181, it does shed some light on the internal workings of ALUs in general.



## 74LS181

With the virtual ALU complete, it is time to turn the focus to the 74LS181. This is a 4-bit ALU from the 1970s capable of 32 different arithmetic and logical operations. A pinout is shown to the right. Inputs S0 to S3 are the select inputs to the ALU, which along with the Mode Control pin decide what operation the ALU will perform. Inputs A0 to A3 and B0 to B3 make up the A and B inputs respectively, and the output of the operation is shown on the F0 to F3 pins. This ALU has only one "flag", the A = B pin which is on when outputs F0 to F3 are on, and this pin is an open collector meaning it must be pulled up by an external resistor. While not necessarily intended as a flag, the $C_{n+4}$ pin can be used as a regular carry flag.

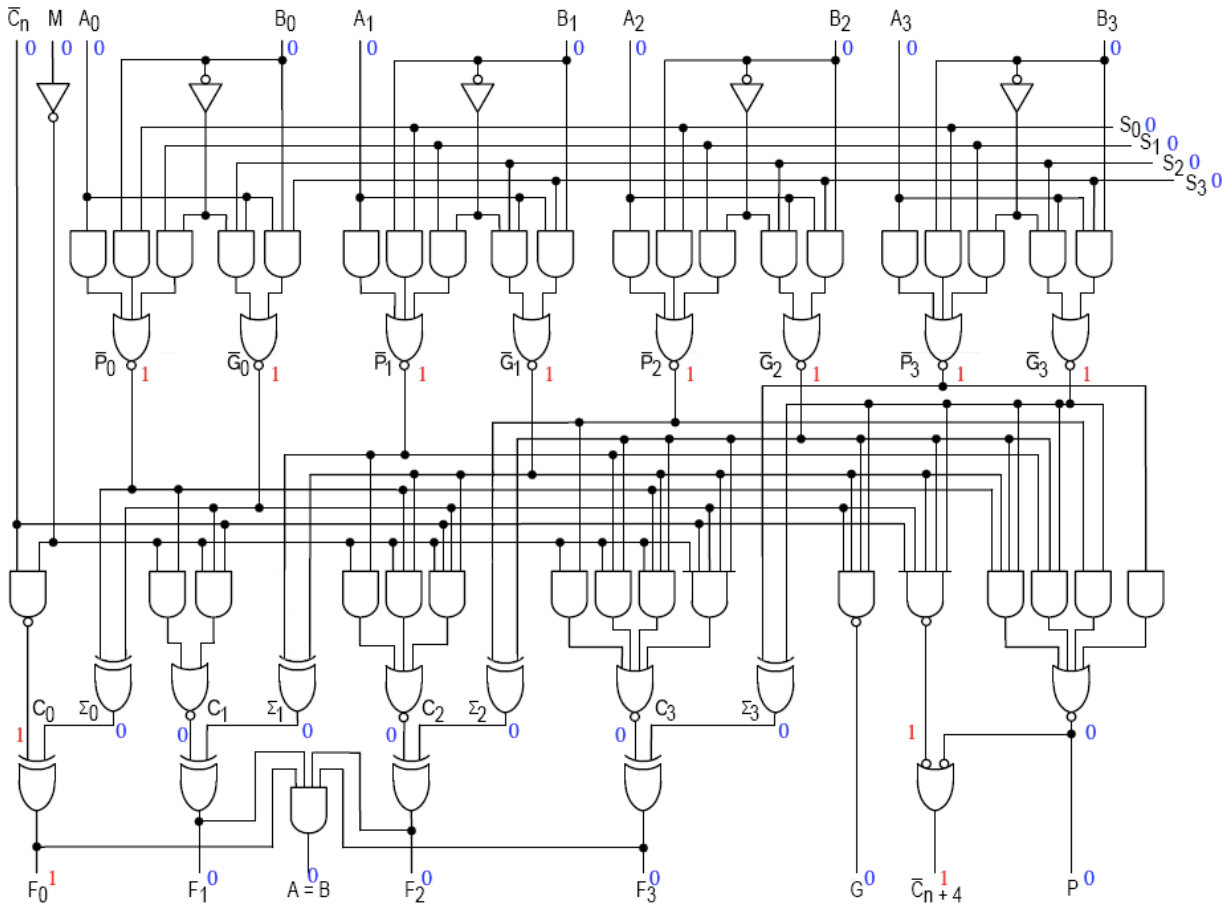



Below is a table of the different arithmetic and logical operations that the 74LS181 can perform. Note that the mode-select pin switches between logical and arithmetic operations. The carry pin will add one to the result when performing an arithmetic operation.

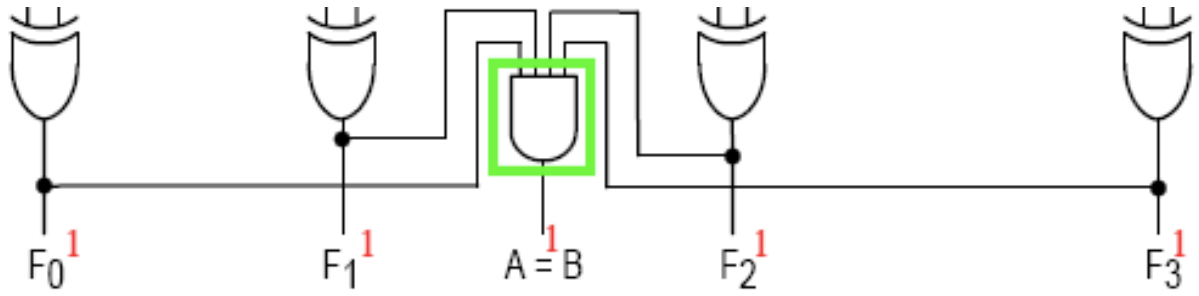
| SELECTION | | | | ACTIVE-HIGH DATA | | |
|---|---|---|---|---|---|---|
| | | | | M = H | M = L; ARITHMETIC OPERATIONS | |
| S3 | S2 | S1 | S0 | LOGIC FUNCTIONS | $\overline{C_n}$ = H (no carry) | $\overline{C_n}$ = L (with carry) |
| L | L | L | L | F = $\overline{A}$ | F = A | F = A PLUS 1 |
| L | L | L | H | F = $\overline{A+B}$ | F = A + B | F = (A + B) PLUS 1 |
| L | L | H | L | F = $\overline{A}B$ | F = A + $\overline{B}$ | F = (A + $\overline{B}$) PLUS 1 |
| L | L | H | H | F = 0 | F = MINUS 1 (2's COMPL) | F = ZERO |
| L | H | L | L | F = $\overline{AB}$ | F = A PLUS A$\overline{B}$ | F = A PLUS A$\overline{B}$ PLUS 1 |
| L | H | L | H | F = $\overline{B}$ | F = (A + B) PLUS A$\overline{B}$ | F = (A + B) PLUS A$\overline{B}$ PLUS 1 |
| L | H | H | L | F = A ⊕ B | F = A MINUS B MINUS 1 | F = A MINUS B |
| L | H | H | H | F = A$\overline{B}$ | F = A$\overline{B}$ MINUS 1 | F = A$\overline{B}$ |
| H | L | L | L | F = $\overline{A}$ + B | F = A PLUS AB | F = A PLUS AB PLUS 1 |
| H | L | L | H | F = $\overline{A \oplus B}$ | F = A PLUS B | F = A PLUS B PLUS 1 |
| H | L | H | L | F = B | F = (A + $\overline{B}$) PLUS AB | F = (A + $\overline{B}$) PLUS AB PLUS 1 |
| H | L | H | H | F = AB | F = AB MINUS 1 | F = AB |
| H | H | L | L | F = 1 | F = A PLUS A | F = A PLUS A PLUS 1 |
| H | H | L | H | F = A + $\overline{B}$ | F = (A + B) PLUS A | F = (A + B) PLUS A PLUS 1 |
| H | H | H | L | F = A + B | F = (A + $\overline{B}$) PLUS A | F = (A + $\overline{B}$) PLUS A PLUS 1 |
| H | H | H | H | F = A | F = A MINUS 1 | F = A |

In the final CHUMP build the ALU will play another important role in addition to performing operations; the ALU will be used for jumping instructions. In the first part of the CHUMP project, the program counter was wired to a NAND gate in order to change the current program address. One of the inputs to this NAND gate will come from a control line, but the other will come from the A = B pin on the ALU. Therefore, it is important to understand the functioning of this pin very well, in addition to how to manipulate it.

To understand that, the full schematic of the 74LS181 is included below. Within it is the secret to understanding how to control the jumping of CHUMP. To see this secret, the only important part of the schematic is the part that contains the F0 to F3 outputs, and the A = B pin.



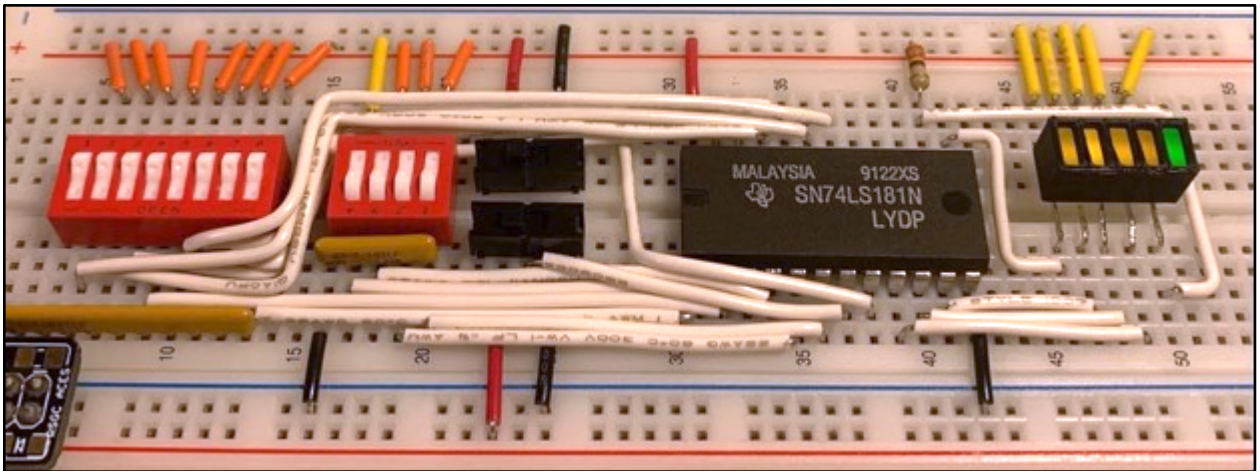

While its name may suggest that the A = B pin is high when input A = input B, the schematic shows that in reality the A = B pin does not care about the A or B inputs, only the F output. The A = B pin is wired through a four-input AND gate from all of the F outputs. Thus, when all of the outputs are high, the A = B pin will be high, and otherwise it will be low.




This means that to manipulate the A = B pins means to control wether or not all F outputs are high. For the GOTO instruction, the A = B pin should be high uncondtionally, so the *Logic 1* operation of the ALU is used, setting all F outputs and high, and so the A = B output follows suit. For IFZERO instuctions, by using the $\overline{A}$ operation, A = B will only be high if the A input is 0.

Below is an image of the circuit for this project which explores the functioning of the 74LS181. The 8-bit switch bank is split into two smaller nibbles, with the high nibble being the A input into the ALU and the low nibble being the B input. The smaller 4-bit switch bank is the S input, and the two single pull double throw switches control the carry and M inputs. The F output is shown on the four yellow LEDs, and the output of the A = B flag is shown on the single green LED.

| Parts Table | |
|---|---|
| 74LS181N ALU | 1 |
| 8-Bit Switch Bank | 1 |
| 4-Bit Switch Bank | 1 |
| SPDT Slide Switch | 2 |
| Bussed Resistor Network | 2 |
| Rectangle 5 mm LEDs | 5 |
| 3D Printed LED Sheath | 1 |



Final ALU Build

## Media

Project Video: https://youtu.be/6YNVlf5IG00

## Reflection

I think that with this project my main struggle was staying on topic. While I really enjoyed bulding a custom ALU from the ground up in Digital Logic Simulator, I admit that it was not exactly within the domain of this project. I included it nonetheless, but if I were to do this project again I would have invested my time elsewhere to take the project further. Despite saying this, I did learn a lot from designing and creating and ALU, and it was an interesting change of pace to design something on the computer where it is simple to completely re-wire something or change connections, which greatly increases prototyping speed compared to breadboarding a circuit.

# Project 3.1.5 CHUMP: Final

## Purpose

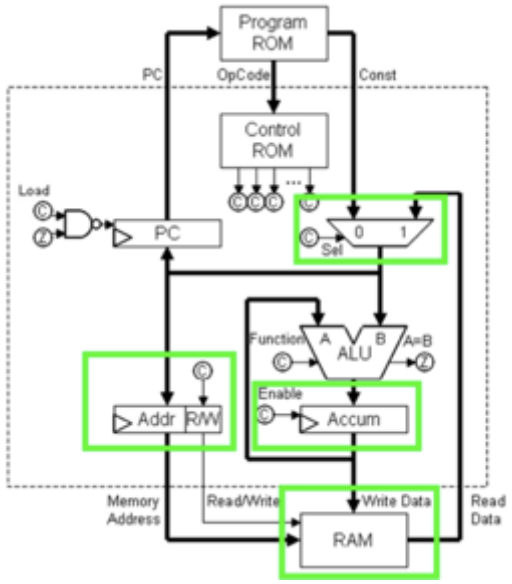
This project marks the final page of the CHUMP build. This stage is where all the different parts of CHUMP will come together and create one cohesive machine able to intelligently receive, process, and execute an instruction set. In addition, this project includes the implementation of an I/O register into the CHUMP build to allow interaction with the user. The addition of I/O to CHUMP teaches about how more sophisticated processors go about adding their own I/O registers and gives a deeper understanding of port and pin manipulation in general.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html
Original Chump Paper: https://www.el-kalam.com/wp-content/uploads/2020/03/A-Simple-and-Affordable-TTL-Processor-for-the-Classroom.pdf

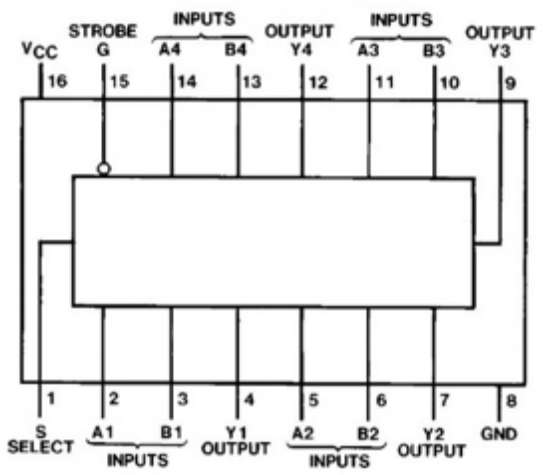
## Procedure

This project introduces four new ICs with new functionalities. These ICs are used for the accumulator, address register, multiplexer, and RAM. The CHUMP architecture diagram to the right shows how these components are included in the processor. Together these four components handle the data and address paths throughout the processor. By taking control inputs from the control EEPROM, these components are collectively responsible for loading values, switching between constant values and memory locations, and reading and writing data to memory. While the rest of CHUMP until now can be thought of as the brains of the processor, this segment contains the actual working parts which are controlled by the brain.



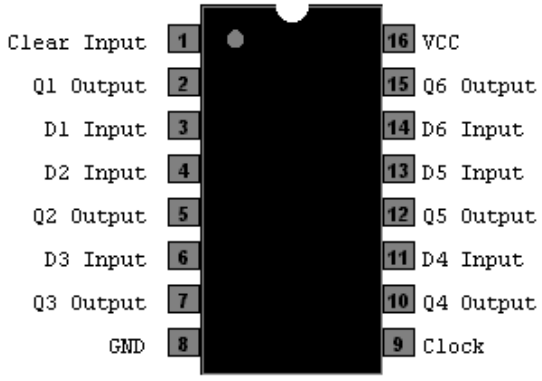

## Multiplexer

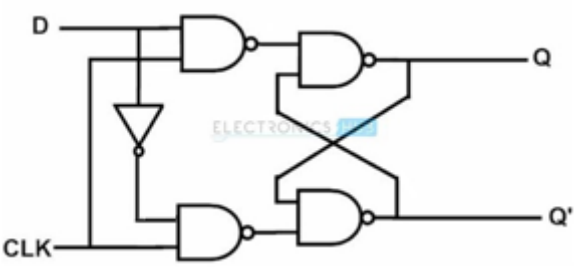
This project is not actually the first use of a multiplexer in CHUMP. In a previous project a multiplexer was used to manipulate two seven-segment displays using POV, but its exact functionality was not explained. In essence, a multiplexer is very simple. The multiplexer used in CHUMP is a 74LS157, which is a 2:1 design. This means that the 74LS157 takes two inputs, and depending on a select signal, outputs either one or the other. In this case, each input is a 4-bit value, with one coming from program EEPROM and the other from RAM. The multiplexer decides which value is used and allows it through to the ALU and address line.



## Address Register

The address register in CHUMP is the 74LS174. This IC is a hex D-flip-flop, meaning it has six D-flip-flop circuits. The address register takes the value out of the multiplexer on its input pins and on the rising edge of the next clock cycle stores that nibble onto its output pins. The address register is accompanied by the R/W bit which comes out of the control EEPROM, and these five bits will then go to RAM to control memory writing and retrieving. The 74LS174 has a clear function on pin 1, meaning pin 1 must be high for normal use as the clear function is active low.

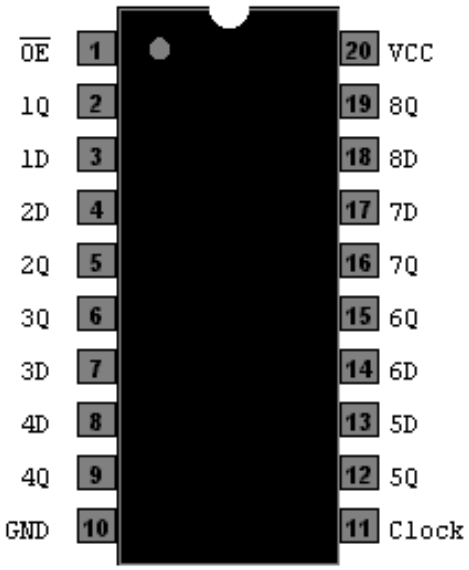



As mentioned, the address register IC is made up of D-flip-flops. While this report has covered D-latches and even edge-triggered D-latches before, the D-flip-flop is a bit different. The D-flip-flop is often used as a basic building block for simple computer memory. The "flip-flop" in the name means that the circuit is triggered on only the rising edge of a clock cycle, where the Q output of the D-flip-flop will mirror input D. That means that every D-flip-flop is an individual bit of edge-triggered memory, so to make up the address register, four D-flip-flops are used, leaving two left over.

## Accumulator

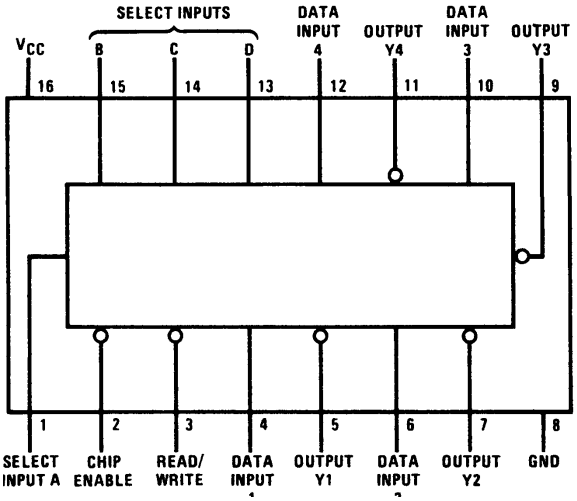
The accumulator is used as a register in CHUMP to store the output of the ALU, and also acts as the ALU's A input. As well as interfacing with the ALU, the accumulator is used to write data to RAM. If the R/W input to RAM is low, signifying a write, the contents of the accumulator will be store to the current address of RAM. The accumulator is a 74LS377, which is quite a similar IC to the address register. This IC is an octal D-flip-flop IC, meaning that the accumulator has eight internal D-flip-flops. However, there is one main difference, being pin 1. This pin is the enable pin on the accumulator, which can be used to turn the chip on and off. This is used to protect the value stored in the accumulator during instructions where the value in the ALU changes but the accumulator is meant to stay the same.




This functionality of preserving the value in the accumulator is what requires the use of this octal flip-flop IC, which would otherwise be overkill. The functionality of the accumulator would be greatly reduced without this pin. For example, GOTO instructions force a high onto all the ALU output pins, which means that without the disable pin for the accumulator it would lose its value on every GOTO instruction (actually on every successful jump).

## RAM

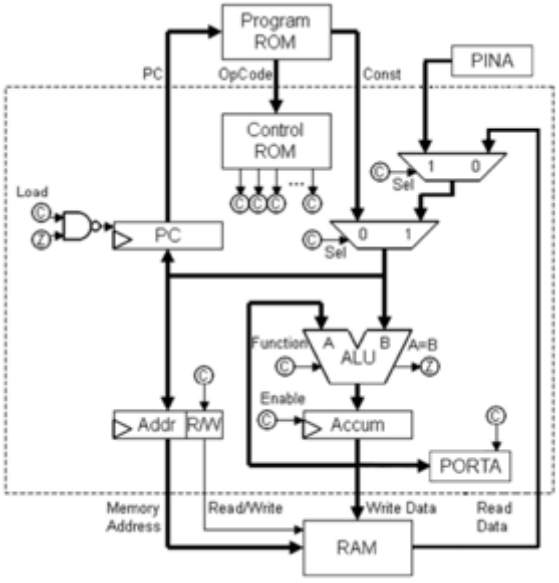
The RAM in CHUMP is used as its long-term memory. Values can be computed using the ALU and accumulator, but if the user wants to save the data for a longer-term, they must store it in RAM rather than the accumulator. While the accumulator is constantly written over, the RAM is kept until CHUMP is unplugged. The IC used for RAM in CHUMP is the 74LS189, shown on the right. Data is written to RAM by setting the R/W input low. The data then presented on the data inputs will be stored to the address presented to the address inputs. To read from RAM simply set the R/W pin high and the output will be shown on the output pins. However, it is important to note that this IC is uses open-collectors for its output pins, so the outputs must be pulled up. This also means that the output of RAM will be the complement of the data stored at that address, so the user must the invert the output (or input) if they wish to read the correct output. This is done with the 74LS04.

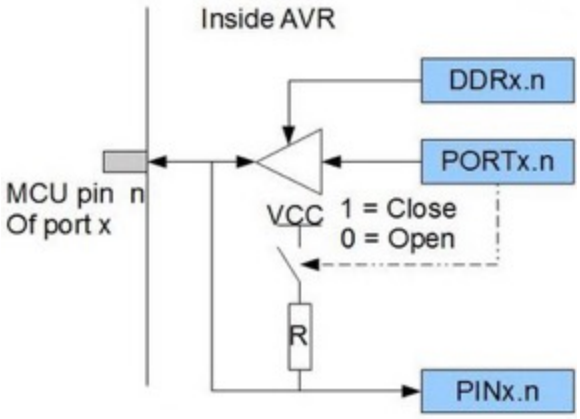



The 74LS189 is a perfectly suited memory IC for CHUMP because of two main reasons. Firstly, it is static RAM, so it does not require refreshing, and secondly, memory its arrangement. The memory of the 74LS189 is arranged into 16 nibbles for a total of 64 bits of memory. CHUMP, with 4-bit address and data busses, is only capable of writing to 16 addresses in the first place, and is only capable of writing 4-bits of data to that address. Therefore, the 74LS189 suits the needs of CHUMP as it perfectly fits its limitations.

## I/O Implementation

As mentioned earlier, this CHUMP build comes with some added features. The biggest change between the standard CHUMP and this one is the I/O register incorporated into the architecture. To the right is a complete CHUMP architecture diagram with I/O implemented. There are quite a few big changes to the standard architecture to implement I/O. Firstly, the I/O comes with two registers, PORTA and PINA. Secondly, because there are now three possible locations to read data from, another multiplexer is required to select between all three paths (a single 3:1 multiplexer would work as well). Thirdly, because the design now requires more control signals, a second control EEPROM was implemented to trigger on the two possible I/O instructions. The design of this new architecture is intentionally unintrusive to the standard CHUMP, meaning all programs that work on the standard CHUMP will still run on this new CHUMP (with one small limitation which will be discussed later).



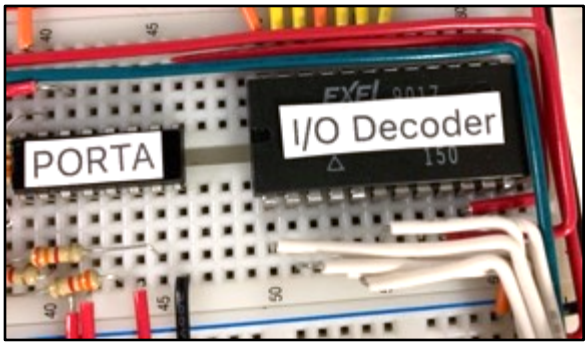



Before getting too far into the I/O of CHUMP it is a good idea to look at how I/O is normally accomplished. While all processors and microcontrollers have different names for the technical details of I/O, this description will stick to the AVR terminology for familiarity. In general, I/O pins on processors are split into ports. These ports will have the same number of I/O pins as the width of the data bus for the processor. This is because when writing to an I/O pin the processor will actually write to a PORT register that controls the states of all pins in the port. However, even before the pins can be written to, they must be declared as either an input or output. This is normally done through a data direction register, or DDR. By writing a 1 to bit *n* of a DDR, pin *n* will then be an output. A 0 normally represents input. Assuming bit *n* in the DDR is 1, the output state of pin *n* will reflect bit *n* in the PORT register. In the case of most processors, writing a 0 to DDR and a 0 to PORT for pin *n* will set pin *n* to a floating state, but this is beyond what is needed for this simple demonstration of I/O. To read from pin *n*, the processor will read from bit n of the PIN register/buffer, which simply reflects the digital value of all the pins in the register.

**CHUMP I/O Configuration**

| PINA | I/O Pin | PORTA |
|---|---|---|
| 0/1 | 0/1 | 0/1 |
| 0/1 | 0/1 | 0/1 |
| 0/1 | 0/1 | 0/1 |
| 0/1 | 0/1 | 0/1 |

To restate, this is the specific implementation of I/O within AVR processors, but it would work perfectly fine in CHUMP and is a good choice due to familiarity with its workings. However, it is a bit overly-complicated for CHUMP. Instead of copy-and-pasting AVR's I/O system, this CHUMP implementation will simplify it down. Firstly, DDR is not strictly necessary without tri-state. The PORT register can act as both PORT and DDR as long as the pin will output a low when PORT is 0. This can be accomplished by a single resistor between the pin and the PORT register, which will pull the pin either up or down. This means that when bit *n* in PORT is 0 (*n* is an input), PIN will read 0 by default, but 1 if there is a high on the I/O pin.




With I/O circuitry figured out, there still needs to be software commands to call the I/O pins into action. That gives two options: 1) two of the existing instructions could be replaced by an instruction to write to PORT and an instruction to read from PIN, or 2), a second control EEPROM can be added to catch two specific full 8-bit instructions out of the possible 256 and use them to control the added multiplexer and PORT clock pin. This second option requires more work and wiring, but is less intrusive to CHUMP build, so it was implemented. A picture of this EEPROM is shown to the right. The full byte from program EEPROM is decoded, and if either of the two I/O instructions are found, the control lines (the two red wires in the top of the picture) are configured accordingly.

The final step of implementing I/O is choosing which two full 8-bit instructions should be used to write to PORT and read from PIN. The low nibble of the instruction is arbitrary, so to for simplicity it will be 0xF. To find the instruction that best suits writing to PORT and reading from PIN it is a good idea to think about what each instruction already does. For example, STORETO already takes the value in the accumulator and stores it, and with the multiplexer setup the instruction must be a memory instruction, so for writing to PORT the clock pin on the PORT register will be set to high when the current instruction is 0x7F (STORETO memory F). For reading from PIN, LOAD already makes a lot of sense. By simply taking LOAD memory 1 (0x1F) and setting the secondary multiplexer's select bit high, the value on PIN will be sent to the accumulator where it can then be manipulated in only a single clock cycle.

## Other Changes

As mentioned in a previous project, the code for this CHUMP build requires a custom instruction, IFNEGATIVE. This has been implemented into the build be eliminating the carry control signal and hardwiring the carry pin of the ALU to the S0 pin, which has the same sequence. This freed up a control line, which was made into the jump control line. All jump instructions now set this line high instead of ANDing the highest two bits of the op-code. This allows for a jump to be executed if the Z or C flag of the ALU is high and the jump line is high (in Boolean logic: (Z+B)J). Below are the new control signals:

| Instruction | S3 | S2 | S1 | S0 | M | Jump | Acc | R/W |
|---|---|---|---|---|---|---|---|---|
| LOAD c | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| LOAD it | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| ADD c | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| ADD it | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| SUB c | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| SUB it | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| STORE c | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| STORE it | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| READ c | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| READ it | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| IFNEG c | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| IFNEG it | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| GOTO c | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| GOTO it | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| IFZERO c | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| IFZERO it | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

The other change has been the code. With the implementation of I/O the program can be made to listen to user input. The inability of CHUMP to do this was actually one of the greatest limitations of CHUMP according to David Feinberg in his paper. The updated code is below, which determines if user input B is greater than user input A:

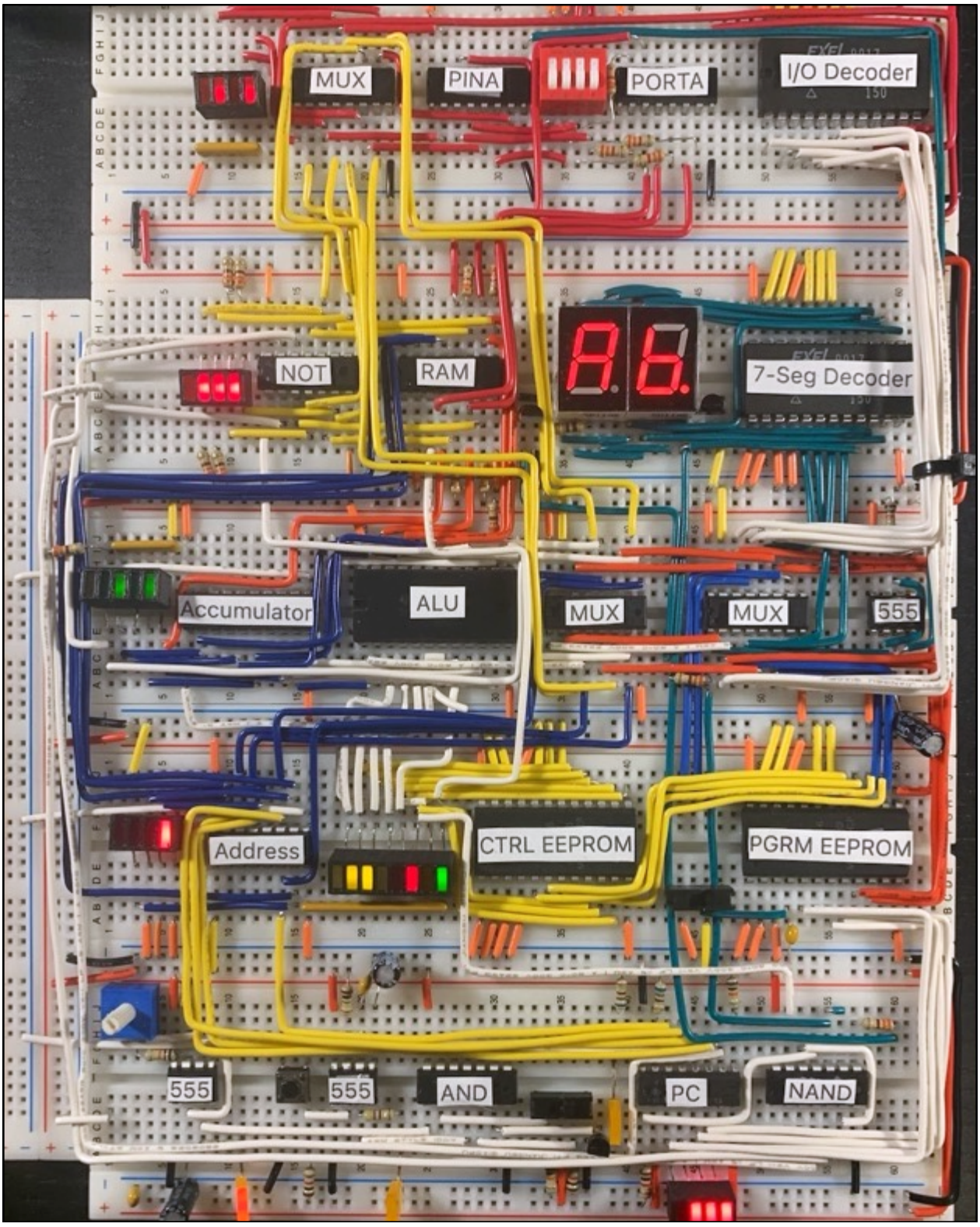
| Line | Machine Code | Hex | Assembly Operation & Operand | High Level |
|---|---|---|---|---|
| 0000 | 00000000 | 0x00 | LOAD 0 | accum ← 0, pc++ |
| 0001 | 01111111 | 0x7F | PORTA | PORTA ← accum, pc++ |
| 0010 | 00011111 | 0x1F | PINA | accum ← PINA, pc++ |
| 0011 | 01100000 | 0x60 | STORETO 0 | mem[0] ← accum, pc++ |
| 0100 | 00011111 | 0x1F | PINA | accum ← PINA, pc++ |
| 0101 | 01100001 | 0x61 | STORE 1 | mem[1] ← accum, pc++ |
| 0110 | 10000000 | 0x80 | READ 0 | addr ← 0, pc++ |
| 0111 | 00010000 | 0x10 | LOAD it | accum ← mem[0], pc++ |
| 1000 | 10000001 | 0x81 | READ 1 | addr ← 1, pc++ |
| 1001 | 10101101 | 0xAD | IFNEG 13 | accum-mem[1] < 0 ? pc=13 : pc++ |
| 1010 | 00000000 | 0x00 | LOAD 0 | accum ← 0, pc++ |
| 1011 | 01100010 | 0x62 | STORETO 2 | mem[2] ← accum, pc++ |
| 1100 | 11000000 | 0xC0 | GOTO 0 | pc ← 0 |
| 1101 | 00000001 | 0x01 | LOAD 1 | accum ← 1, pc++ |
| 1110 | 01100010 | 0x62 | STORETO 2 | mem[2] ← accum, pc++ |
| 1111 | 11000001 | 0xC1 | GOTO 0 | pc ← 0 |

## Final Parts Table

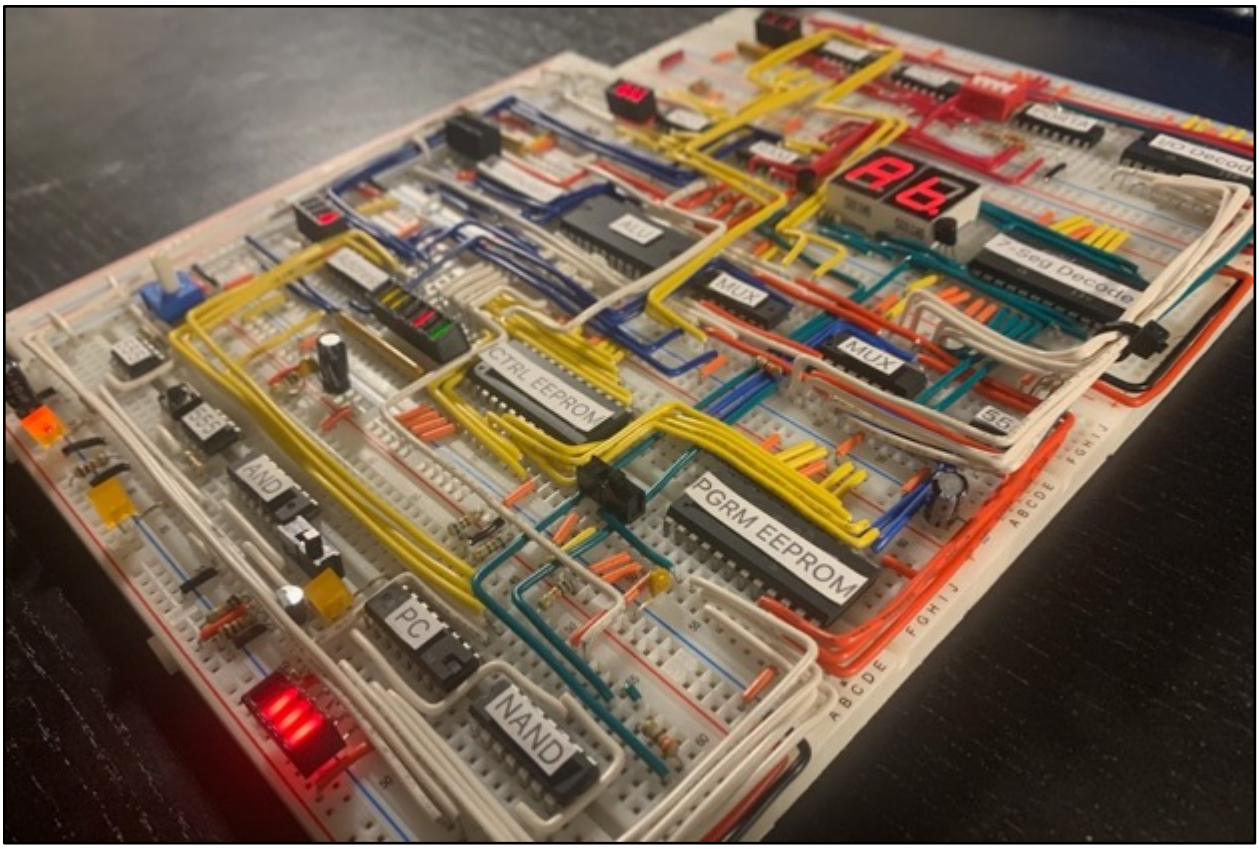
| **Parts Table (Entire CHUMP)** | |
|---|---|
| AT28C16 Parallel EEPROM | 4 |
| 74LS157 Two-Channel Multiplexer | 3 |
| 555 Timer | 2 |
| 74LS189 Static RAM | 1 |
| 74LS174 Hex D-Flip-Flop (Address Register, PORTA, PINA) | 3 |
| 74LS377 Octal D-Flip-Flop (Accumulator) | 1 |
| 74LS04 Hex Inverter | 1 |
| 74LS181 Arithmetic and Logic Unit | 1 |
| 74LS00 NAND Gate | 1 |
| 74LS161 4-Bit (Program) Counter | 1 |
| 3904 NPN BJT Transistor | 2 |
| Common Cathode Seven-Segment Display | 2 |
| SPDT Slide Switch | 2 |
| 3D Printed 8-Bit Bar-Graph Sheath | 6 |
| 4-Bit Switch Bank | 1 |
| Assorted LEDs | ** |
| Assorted Resistors | ** |
| Assorted Capacitors | ** |

## Media

Project Video: https://youtu.be/DcIRDraD-Dk




Final CHUMP Build




CHUMP Side View

## Reflection

This project was a ton of fun to build. The room for customisation within CHUMP is enormous, and gives room to branch out into anything you would like. No two CHUMPs will end up looking alike. The other thing I liked about CHUMP is how the group comes together to get it done. We all had our own chip assignments, and we taught and learned with each other. For example, two of us were interested in implementing I/O into CHUMP as we both thought it was the logical next step, and even though we ended up disagreeing on the important criteria for I/O, how to best implement it, and ended up going very different ways with the implementation, the experience of going through the project with other classmates was still quite enjoyable. All that to say, I am happy that CHUMP is now complete and I think I will be taking a break from wiring for some time.

# Project 2.10c Short ISP: DIY Variable Power Supply

## Theory

This project is very theory-intensive. The overall function of a variable power supply is to take the 120 V, which is AC, out of the wall and somehow turn that into a variable low voltage DC output. This is a daunting task to look at as a whole, but it can be broken down into four manageable sized pieces that individually are possible to tackle:

1. The first big idea in this project is the idea of stepping down AC voltages. The 120 V AC from the wall is obviously much too high to power any circuits even if it were DC. To take this AC voltage and step it down, which simply means to decrease it to some lower value, this project employs a transformer, which is an incredibly versatile and interesting component.

2. Secondly, with this low voltage AC signal, the circuit then has to transform this into a DC signal. To accomplish this, a full bridge rectifier is used which is a common way to essentially take the "absolute value" of an AC waveform and convert it to a DC waveform.

3. Next, the circuit must have a way to alter this low voltage DC signal, as in the name of a variable power supply. In this case a low-side buck converter in used, which is a less common variation of the normal high-side buck converter which can be thought of as a method of varying voltage similar to PWM on microcontrollers.

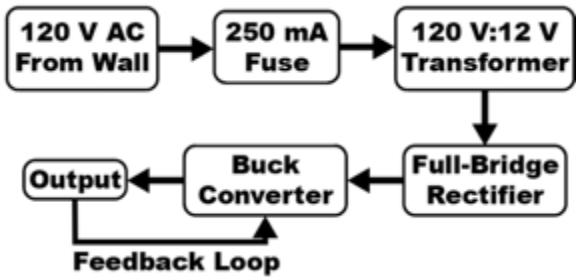
4. Finally, the circuit must have a feedback loop to ensure that changing current draws from the load across the output do not alter the voltage presented by the circuit. To do this, this project uses a custom op-amp feedback system for low-side buck converters, which is typically avoided for a few reasons that will be discussed later.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/2324/ISPs.html
Low-Side Buck Converter Topology: https://electronics.stackexchange.com/questions/330471/low-side-n-mosfet-buck-converter
ATtiny85 Datasheet: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf

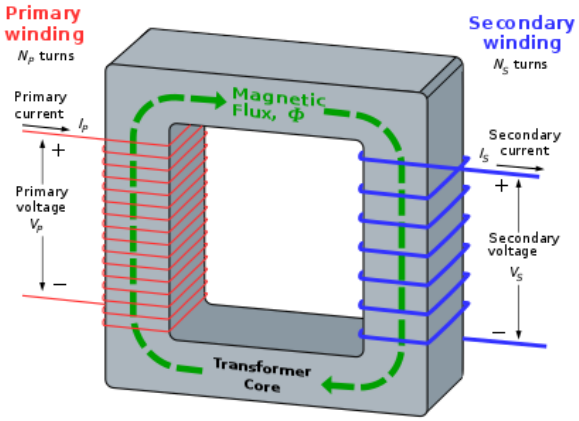
## Procedure

To the right is admittedly an incredibly simplified diagram of how this power supply functions, but the basic flow diagram has a lot to offer without going any deeper into each component. Some parts of this diagram are quite simple, and others are very intricate, so each will have its own section with an explanation of its function, how it relates with the rest of the build, and how it works.



## Part 1: The Transformer

Discounting a fuse, the transformer in any appliance is the first component in the circuit. A transformer is a general term for any circuit with at least two windings of wire around a core where an alternating voltage in one winding (the primary) induces a voltage across the other winding (the secondary). This voltage is induced through the magnetic field in the core, which is typically a material with a high relative permeability to achieve a high magnetic flux density in the core. This alternating magnetic field in the core is created by the alternating electric field of the primary which in turn causes an alternating electric field in the secondary as well.




The number of turns in each winding relates the current and voltage ratio of each. In general, if the secondary has less turns than the primary, it will have a lower voltage across it but a higher maximum current, and if it has more turns it will have a higher voltage than the primary but a lower potential current draw. Note that this ratio assumes that there are plenty of turns in each winding. For example, two turns of a primary and a single turn of a secondary will induce a negligible voltage into the secondary as no core material is ideal; transformers generally require hundreds of turns in each winding.

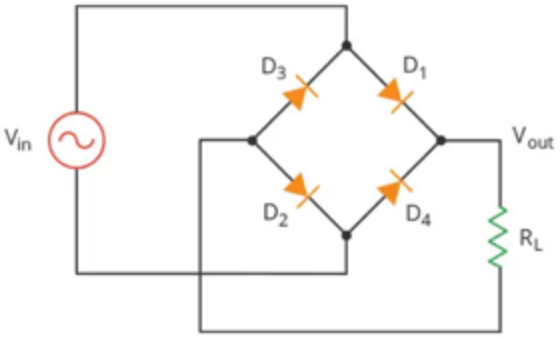
This turn ratio is direct, meaning if the secondary has $n$ times the turns as the primary, the voltage across the secondary will be $n$ times the voltage across the primary. Conversely, the current relationship between the windings is inversely relate to $n$, so the current capacity of the secondary will be $n^{-1}$ times the current capacity in the primary. These equations are demonstrated below:

$$\frac{V_1}{N_1} = \frac{V_2}{N_2} \quad \text{and} \quad \frac{I_1}{I_2} = \frac{N_2}{N_1}$$

This means that the transformer used in this supply which has a 120 V AC input and 12 V AV output has a primary with 10 times the turns as in the secondary, and theoretically means the maximum current through the secondary is 100 amps (the maximum power of AC mains is 10 amps at 120 V) resulting in a maximum power of 1200 W. This calculation exposes the physical limits of transformers, as the actual maximum power rating of this transformer is a mere 10 W. Transformers are generally limited by the ampacity of the wire used in their windings. Wires in transformers are generally quite thin to allow many turns to fit into a small form factor, which limits the current that they can source.

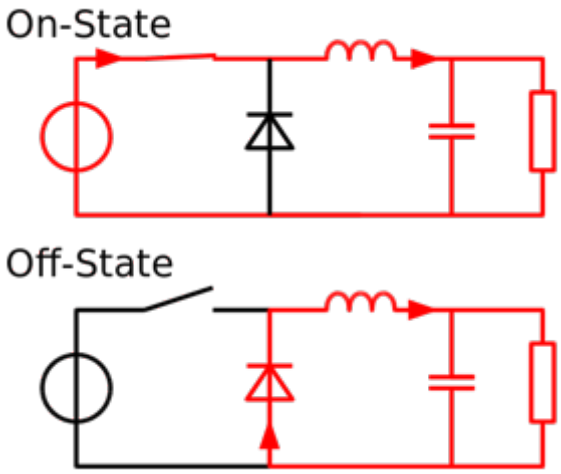
## Part 2: The Full-Bridge Rectifier

After the transformer steps down the 120 V AC to a useable 12 V AC the signal must be rectified, which means it must be turned into DC. This is done by the full-bridge rectifier, which will always allow the high side of the AC waveform to pass through as a DC signal. After this the signal can then be smoothed by capacitors to generate a stable 12 V DC for the circuit to use. Note that the AC signal will drop voltage as it passes through the diodes, and for this reason many transformers slightly overperform on output voltage in anticipation of being rectified.



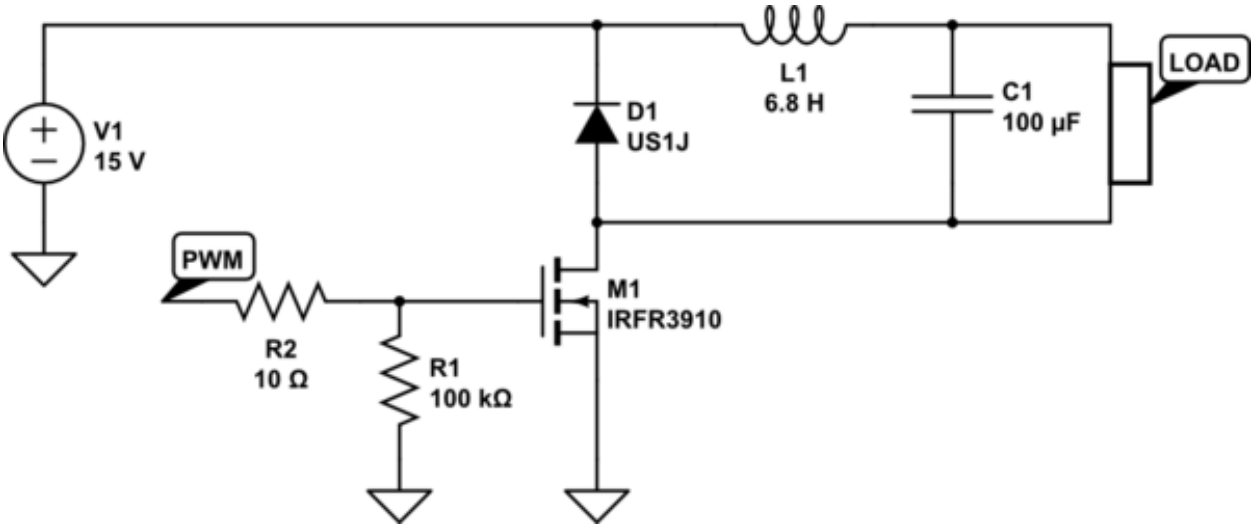
## Part 3: The Buck Converter

The buck converter is the part of the circuit where the output voltage is varied. Buck converters are DC to DC step down converters, meaning they are circuits that can output a lower voltage than their input source voltage.




To the right is the topology of a typical high-side buck converter. While this project uses a low-side topology, most of how the buck converter actually functions is the same. To begin with, the output voltage across the load is controlled by varying the duty cycle of the switch shown in the diagram. When the switch is closed for the first time, the voltage across the load approaches the source voltage of the circuit while the capacitor and inductor charge. When the switch is open, the voltage across the load begins to decrease as the inductor draws energy from its magnetic field and the capacitor draws from its electric field to attempt to sustain the current and voltage across the load. This is because inductors resist changes in current, and capacitors resist changes in voltage. When the switch is closed again, the inductor and capacitor once again resist a change in current and voltage, maintaining smooth conditions across the load. Overall, the inductor and capacitor work to smooth out the PWM signal generated by the switch.

In this project a low-side topology is used, as shown below. This topology is chosen for practicality. While in the diagrams above the circuit is controlled by a switch, in the real world a MOSFET is generally used, and high side switching of MOSFETS requires complex driving circuits and negative voltages, both of which can be avoided by moving to a low side N-channel MOSFET topology as shown below.




There is one major difference between the two designs, which is why high-side P-channel buck converters are still useful. In the traditional high-side design, the positive terminal of the output is scaled depending on the PWM while the ground is the same ground as the driving circuit. However, in low-side topologies, the positive terminal of the output is fixed to the supply voltage of the driving circuit while the ground terminal of the output varies, which makes it difficult to read the voltage across the output for feedback.

The final point of discussion for the buck converter is the method of providing a PWM signal to the MOSFET. In this circuit, an ATtiny85 is used for variable PWM purposes. Normally the PWM of the ATtiny85 sits are around 500 Hz, but for buck converters this is much to slow. However, by editing the timer control registers of the timer we are using for PWM, we can remove the prescaler which slows down the PWM frequency. By doing this, the frequency can get as high as 60 kHz, which is suitable for a buck converter. This means that while this project does technically have a microcontroller in it, it is not really microcontroller controlled as it is only used for PWM.

To know which bits to set in which registers it is good to look at the ATtiny85's datasheet from Atmel. Which timer is used does not matter all that much, so TIMER0 was chosen. According to the datasheet, TIMER0 has two control registers, TCCR0A and TCCR0B (TCCR stands for timer/counter control register):

**TCCR0A – Timer/Counter Control Register A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x2A | COM0A1 | COM0A0 | COM0B1 | COM0B0 | – | – | WGM01 | WGM00 | TCCR0A |
| Read/Write | R/W | R/W | R/W | R/W | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**TCCR0B – Timer/Counter Control Register B**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| 0x33 | FOC0A | FOC0B | – | – | WGM02 | CS02 | CS01 | CS00 | TCCR0B |
| Read/Write | W | W | R | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The next step is to look at what each of these bits do. According to the datasheet, the important bits for fast PWM seem to be COM0A1, COM0B1, WGM01 and WGM00 in TCCR0A and all three CS0 bits in TCCR0B. COM0A1 and COM0B1configure the output of TIMER0 to be non-inverting when high, so they should both be high.

**Table 11-5.** Waveform Generation Mode Bit Description

| Mode | WGM 02 | WGM 01 | WGM 00 | Timer/Counter Mode of Operation | TOP | Update of OCRx at | TOV Flag Set on |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Normal | 0xFF | Immediate | MAX(1) |
| 1 | 0 | 0 | 1 | PWM, Phase Correct | 0xFF | TOP | BOTTOM(2) |
| 2 | 0 | 1 | 0 | CTC | OCRA | Immediate | MAX(1) |
| 3 | 0 | 1 | 1 | Fast PWM | 0xFF | BOTTOM(2) | MAX(1) |
| 4 | 1 | 0 | 0 | Reserved | – | – | – |
| 5 | 1 | 0 | 1 | PWM, Phase Correct | OCRA | TOP | BOTTOM(2) |
| 6 | 1 | 1 | 0 | Reserved | – | – | – |
| 7 | 1 | 1 | 1 | Fast PWM | OCRA | BOTTOM(2) | TOP |

As shown above, when WGM01 and WGM00 are high, the timer is set to fast PWM, so those bits will both be high.

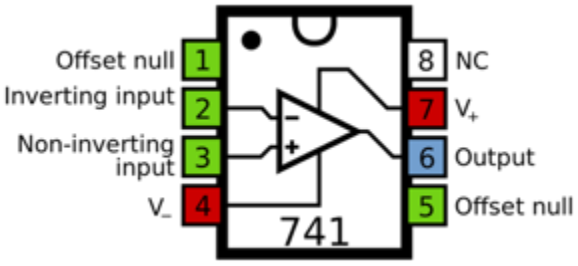
In TCCR0B only the CS bits are important, which are the bits that select the prescaler. The prescaler is essentially a modifier that will divide the source frequency of the timer on the output, slowing down the PWM, so CS02 and CS01 are set to 0 and CS00 is set to 1 to select no prescaling.

| CS02 | CS01 | CS00 | Description |
|---|---|---|---|
| 0 | 0 | 0 | No clock source (Timer/Counter stopped) |
| 0 | 0 | 1 | $clk_{I/O}$/(No prescaling) |
| 0 | 1 | 0 | $clk_{I/O}$/8 (From prescaler) |
| 0 | 1 | 1 | $clk_{I/O}$/64 (From prescaler) |
| 1 | 0 | 0 | $clk_{I/O}$/256 (From prescaler) |
| 1 | 0 | 1 | $clk_{I/O}$/1024 (From prescaler) |
| 1 | 1 | 0 | External clock source on T0 pin. Clock on falling edge. |
| 1 | 1 | 1 | External clock source on T0 pin. Clock on rising edge. |

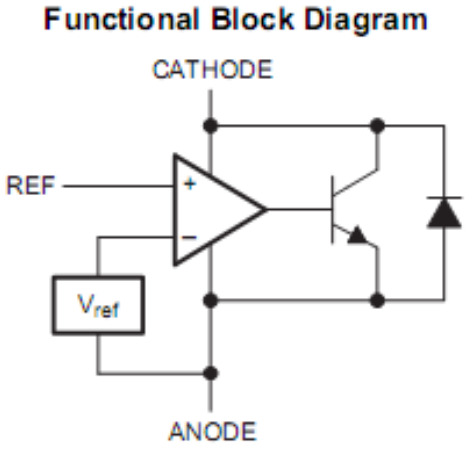
## Part 4: Feedback Loop

The buck converter is not truly complete without feedback. By itself the buck converter will create a varying output voltage, but if the resistance or current draw of the load changes so will the voltage across the load. To keep this voltage stable, a simple feedback cycle is introduced. While the output voltage of the buck converter cannot be directly measured, the ground terminal out the output can be measured and is still proportional to the voltage across the output. Thus, by comparing the voltage from the supply ground to the output ground with a reference voltage the circuit can generate an error signal.

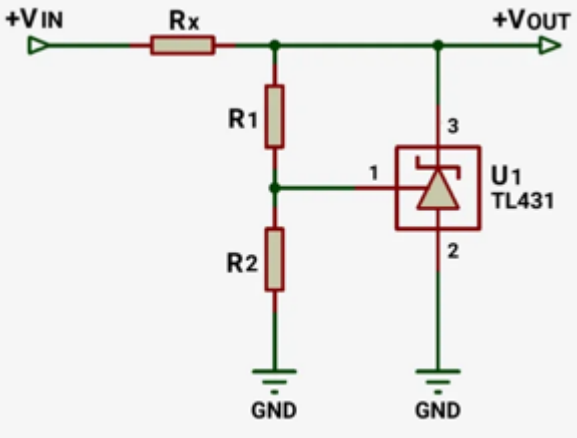
To do this, an LM741 op-amp is used. On the non-inverting input is the reference voltage, and on the inverting input is the ground of the buck converter's output. If the comparison's output is high, that means the output voltage of the buck converter is too high, so the ATtiny85 will read this signal and lower its PWM duty cycle. Conversely, when the comparison's output is 0 the output of the buck converter is too low, so the microcontroller increases its PWM duty cycle.




The reference voltage for the op-amp is generated by a TL431, which is sometimes called a variable Zener diode. Internally the TL431 looks like the schematic to the right. It is an IC made specifically for generating stable reference voltages as it has an internal feedback system to make sure the output voltage is constant. This is why it was chosen as a reference voltage, as components such as potentiometers would have changed their reference voltage as the current draw from the op-amp's input changed slightly over time, making the TL431 a more reliable reference.

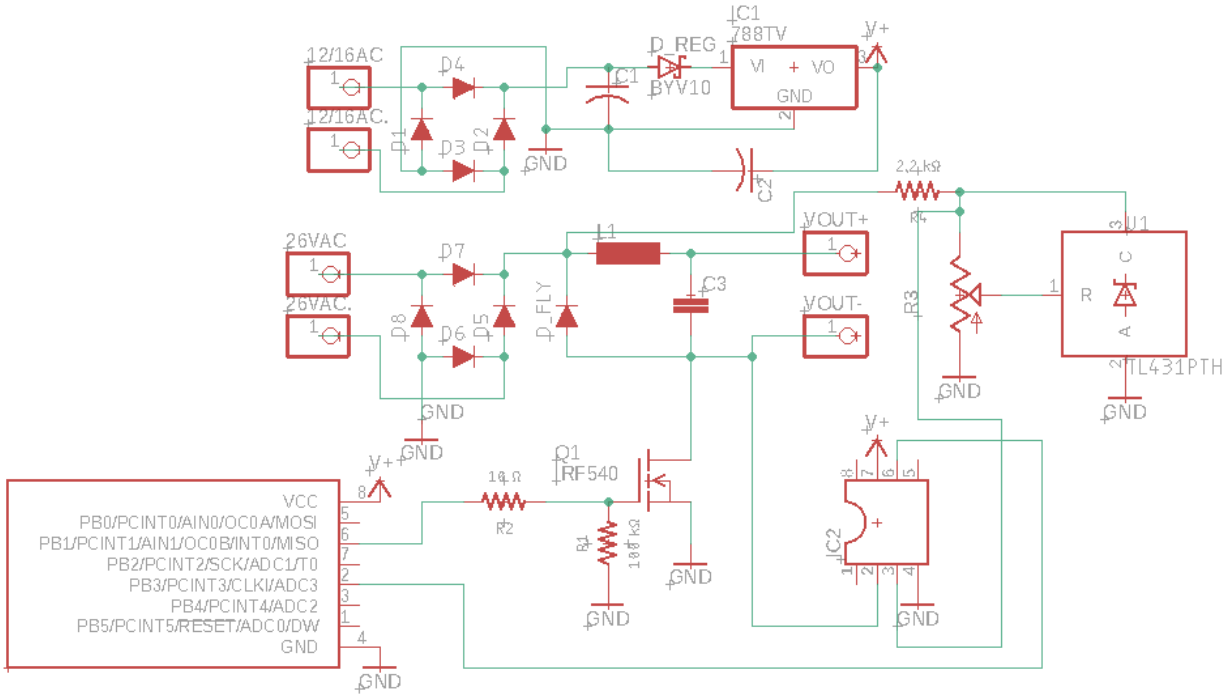


To use the TL431 as a reference voltage there is a very simple circuit which can be constructed with a couple resistors to control the output voltage, as shown to the right. By replacing the two resistors in the voltage divider with a potentiometer with its B lead on pin 1 of the TL431, the output reference voltage will vary with how much the user turns the potentiometer. While having 0 resistance on one side of the potentiometer is not a problem for the TL431, it is important to note that it is not meant to drive any other components. The maximum current source of this IC is very low and it is meant only as a reference voltage, not a variable voltage regulator.




## Dual Power Design

This power supply is designed for a transformer with multiple secondary windings. The supply has two AC inputs into the main board, one for 12 V AC which is rectified and then regulated to 5 V with a 7805 voltage regulator and one for the high voltage output of the board. The two rectified voltages share a common ground reference, and the microcontroller and op-amp run off of the 5 V input while the rest of the circuit deals with the higher voltage source to achieve higher output voltages than the 5 V that the ATtiny85 requires.



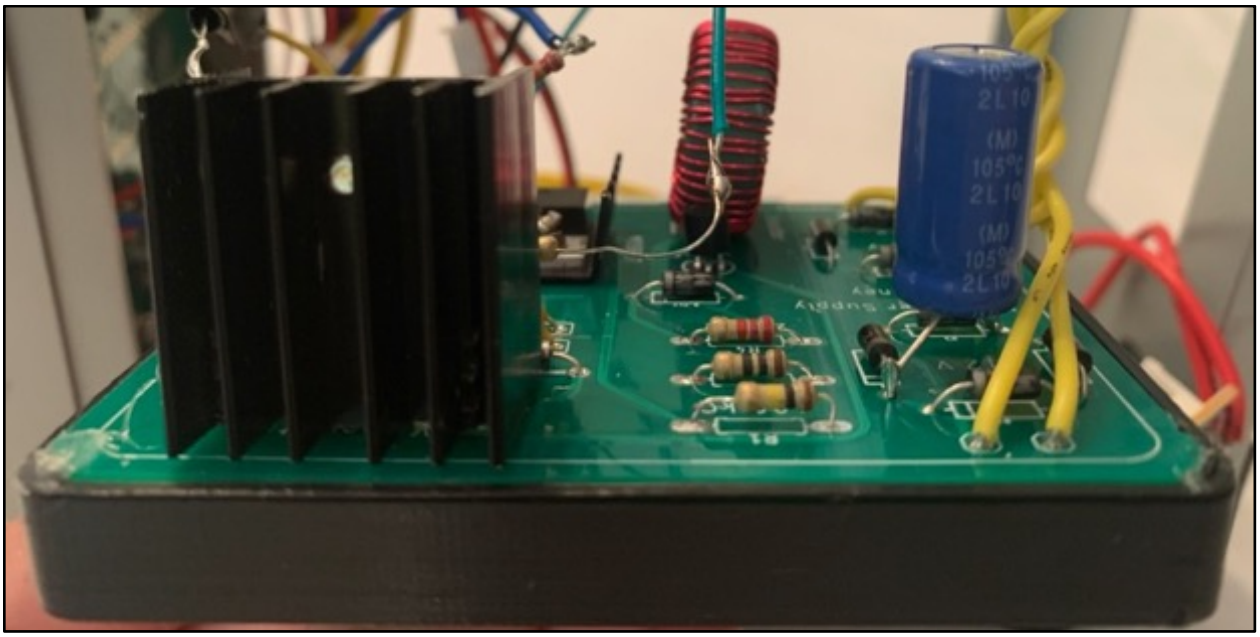
| Parts Table (DIY Power Supply) | |
|---|---|
| 120 V : 12 V Transformer | 1 |
| Custom Buck Converter PCB | 1 |
| IRF540N N-Channel MOSFET | 1 |
| LM7805 Voltage Regulator | 1 |
| Hand-Wound Toroid Inductor | 1 |
| LM741 Op-Amp | 1 |
| ATtiny85 | 1 |
| 1N5817 Diode | 10 |
| TL431 Variable Zener | 1 |
| 0.1 µF Ceramic Capacitor | 3 |
| 250 mA Fuse | 1 |
| Assorted Resistors | 3 |
| Assorted Wires | ** |
| Broken Power Supply Encasement | 1 |

## Code

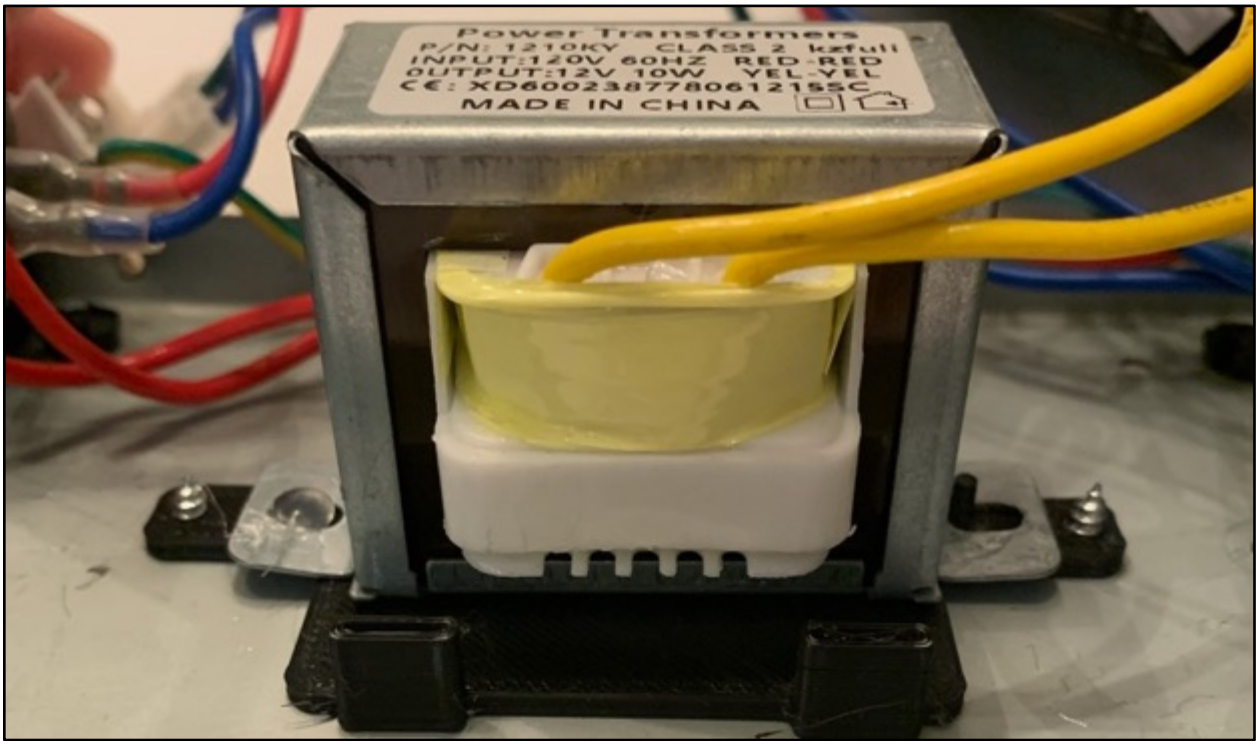
```
//Purpose : Fast PWM for the custom feedback loop of the buck converter
//Date    : Dec 16, 2023
//Author  : Liam McCartney

#define F_CPU 8000000
#define pwm 1
#define feedbackInput A3
uint8_t value = 0;

void setup() {
  TCCR0A = 2 << COM0A0 | 2 << COM0B0 | 3 << WGM00;
  TCCR0B = 1 << CS00;
  /*
  TCCR0A = 2 << 6 | 2 << 4 | 3 << 0;
  //Set the output of TIMER0 to be non-inverting, and set the mode to fast PWM
  TCCR0B = 1 << 0;
  //Select no prescaler for TIMER0

  TCCR0A = 0b10100011;
  TCCR0B = 0b00000001;
  */

  pinMode(pwm, OUTPUT);
  pinMode(A2, OUTPUT);
}

void loop() {
  analogWrite(pwm, value);
  analogRead(feedbackInput) ? value-- : value++;
  value = constrain(value, 1, 254);
  //Alter the PWM duty cycle based off the Op-Amp's output
}
```

## Media

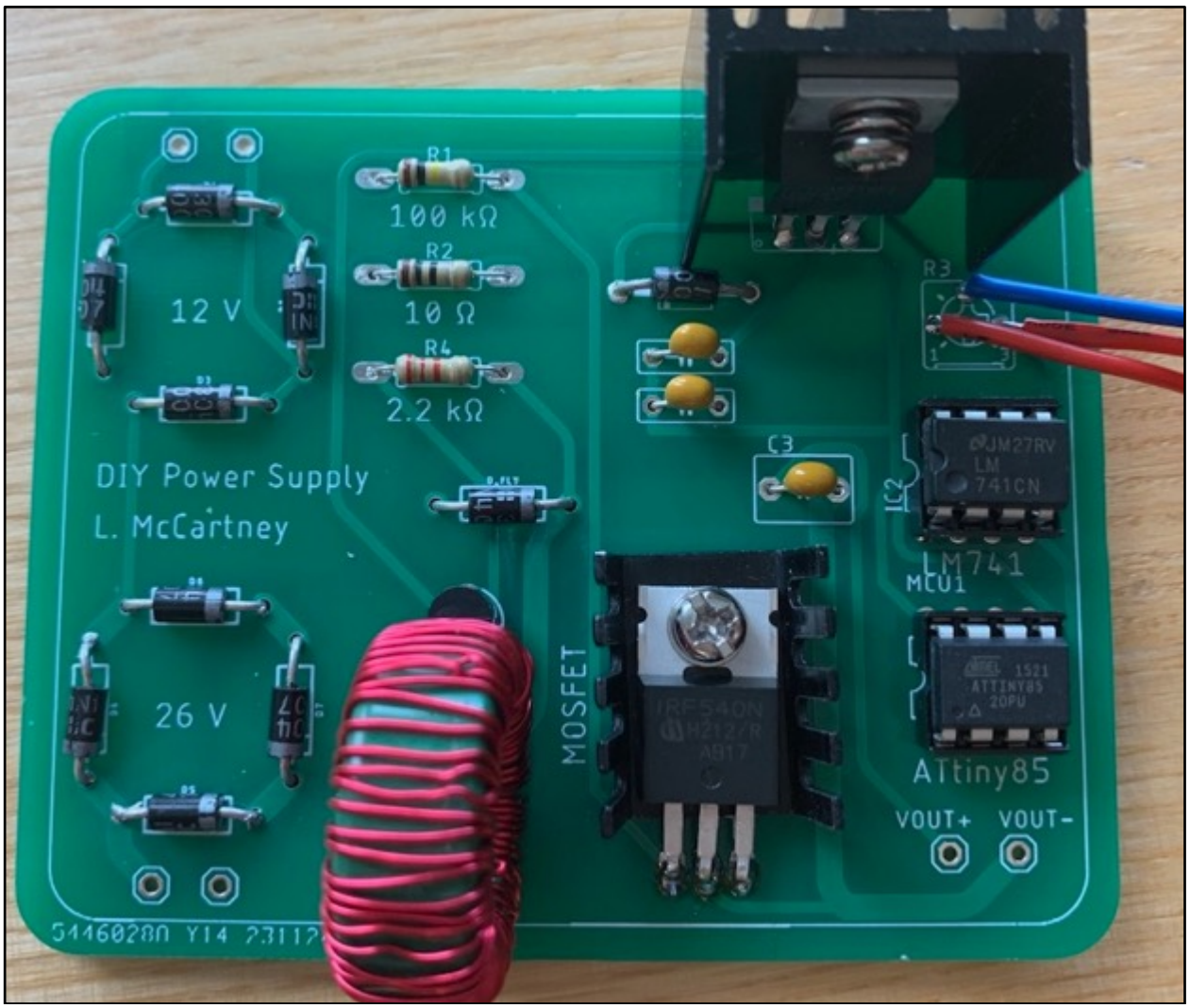
Project Video: https://youtu.be/W-L584rNyy4



PCB in 3D Printed Case




Transformer in 3D Printed Bracket



Buck Converter Soldered PCB

## Reflection

I think that this ISP was a fun dive into electrical engineering that I had never taken before. This was my first ever complex project to not make use of a microcontroller, and I enjoyed that aspect of this project. It seems to me that whenever I use a microcontroller in my build, most of my time ends up being disproportionately spent on writing code rather than on building circuits, designing PCBs, or doing 3D design. While I certainly enjoy coding, I also like the other aspects of Hardware that coding can sometimes overshadow. That being said, after this break from coding I think I will return to a more balanced ISP with software in the new year as I have not challenged myself with software in quite some time, the last difficult piece of code I wrote for a microcontroller was during Data Logger last year.

With this being an ISP I also feel I have to reflect a bit on my time-management throughout the past three months. I gave myself plenty of time to figure out the hardest part of my ISP, the buck converter. There is little information online about feedback systems for low-side buck converters as high-side topologies are more power efficient, and in the industry that means that companies spend the time designing high-side driving circuits with completely different feedback models than with low-side topologies. This meant that I was essentially left on my own to come up with a feedback system, and while I did take a couple wrong turns along the way in the end I am happy with my relatively simple and intuitive design.

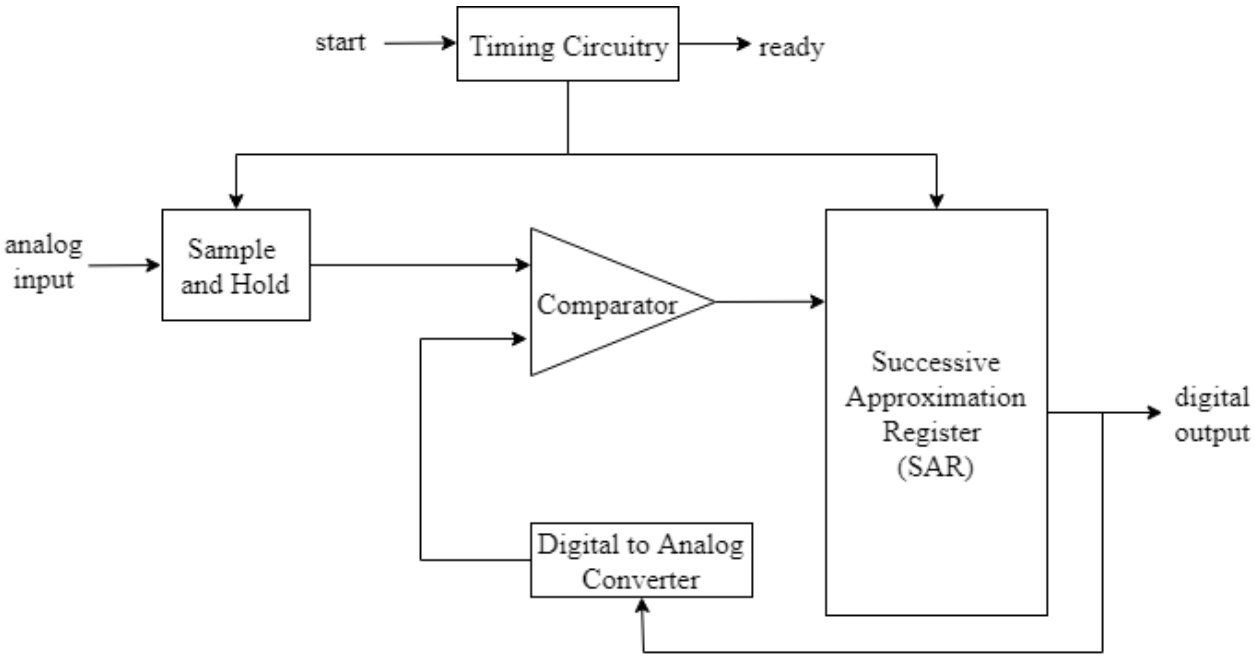
# Project 3.2.1 SAR ADC: Overview and Clock

## Theory

This project is largely theoretical in nature; the most important takeaway is to know the basic function of the SAR ADC that will be built in the coming reports. To begin with, what does SAR ADC even mean? A successive approximation register analog-to-digital converter, or SAR ADC, is a device used to provide a binary (or digital) representation of an analog reference signal. The SAR portion of the SAR ADC refers to the inner workings of the ADC. SAR ADCs use a successive approximation algorithm to essentially "zero in" on the most accurate digital representation of an analog input.

### Successive Approximation Register

The topic of how the successive approximation algorithm of the ADC functions warrants a detailed explanation. Below is a basic flow diagram for a typical SAR ADC:

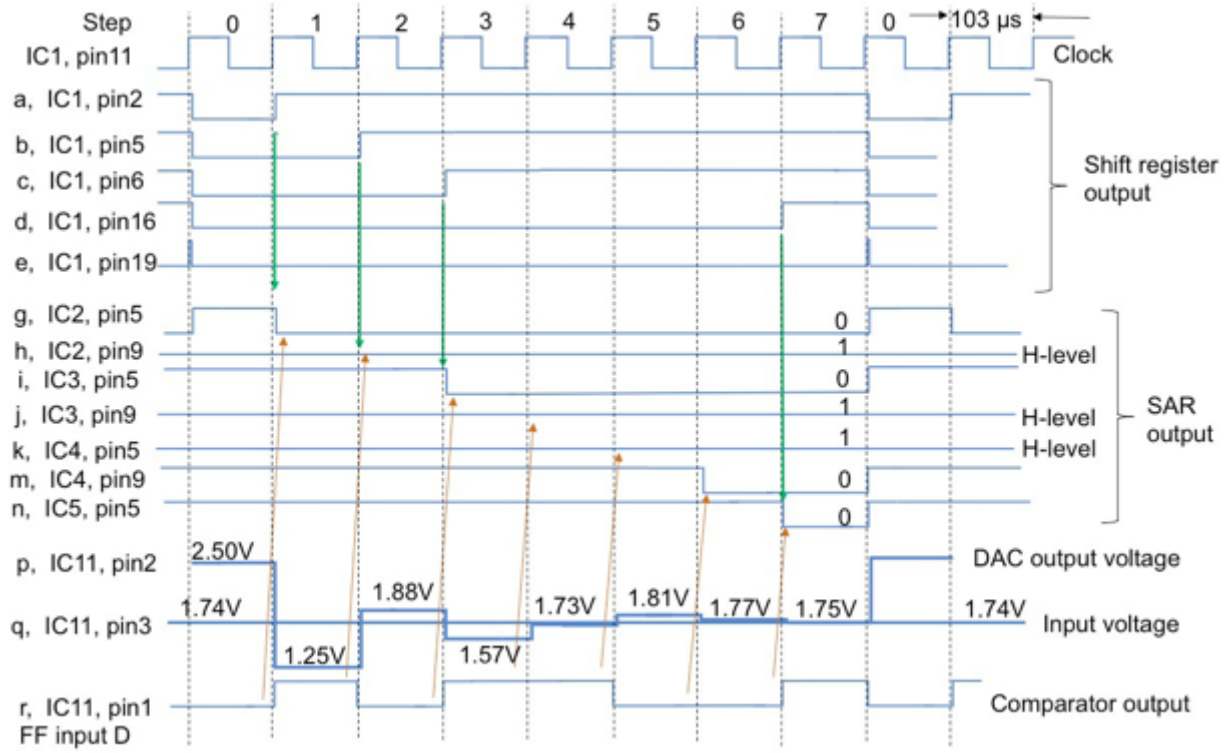



To begin with, the analog input first enters the ADC through the sample and hold circuit. This circuit does as is in the name, it takes the current analog input and preserves it. It takes a new input to sample every eight clock cycles, which is the time taken to perform one conversion, meaning that it holds an input sample taken at the beginning of the conversion for the duration of the process. This is important so as to accurately convert the sample, it is hard to do this if the sample value is constantly changing. This circuit operates by holding the sample signal in a capacitor where it can then be used for comparison.

From this point, the stagnant analog signal enters the portion of the ADC which carries out the successive approximation algorithm. This portion of the circuit (which will be referred to as the SA unit) composed of a comparator, DAC, and the SAR can be thought of as its own unit to simplify the workings of the ADC.

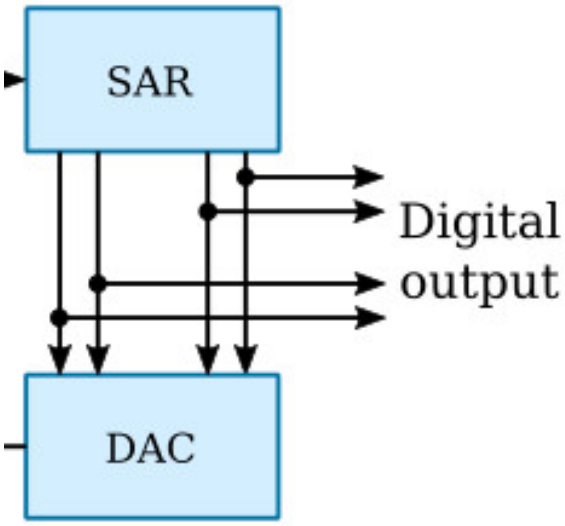
It may at first seem peculiar that DACs (digital to analog converters) are so integral to the function of SAR ADCs. However, once looking how the SA unit works, it quickly becomes apparent. At the start of a conversion, the DAC within the SA unit will generate an analog voltage that is in the middle of the input voltage's range, which for simplicity will be 2.5 V (half of 5 V). This is then compared to the analog sample. Depending on the output of this comparison, the next steps will be different, but either way the output of this comparison is stored in the SAR. If the result is 1, meaning the sample is higher than the DAC test voltage, the DAC will then produce a new test signal in the middle of the upper half of the voltage range, so in the 5 V example this mean 3.75 V. Similarly, if the result is a 0, the DAC will produce a test voltage in the middle of the lower half of the voltage range, so 1.25 V. This comparison cycle is repeated for every bit of information in the ADC, so with the 7-bit ADC in this project this cycle will occur seven times.

This may be hard to imagine, so below is a timing diagram of the ADC taken from Yamada's blog post. In this example it is clear how the DAC is manipulated by the comparator output to produce the next test voltage to compare to the sample.The relavant section is at the bottom of the diagram.




The next most important thing to understand about an SAR ADC is the importance of these comparator signals. As shown by the arrows on the diagram, each clock cycle (besides step 0), the previous comparator output is stored to the coresponding of the SAR. This is why it takes 8 cycles to determine a 7-bit approximation; tick 0 is used to complete the first set comparisson, but no bit can be set within that tick.

The resultant value of the SAR is then interpreted by the DAC. Consider the DAC in its starting state, with its MSB high and all other bits low. At the beginning of the second clock cycle (tick 1), if the MSB bit in the SAR is a high, then the MSB in the DAC will stay high. If not, that bit in the DAC will go low. The next most significant bit in the DAC will always be set high for the next test voltage. This process is repeated every clock cycle, but the bit being scrutinized shifts down in significance every tick. For example, at the beginning of the third clock cycle (tick 2), it is the second MSB which the SAR's output is used to determine. This is often shown in diagrams with a simple connection from the SAR to the DAC, as in the image to the right, and while this works well to illustrate the general idea it is important to know that there is more going on under the hood.




## Purpose

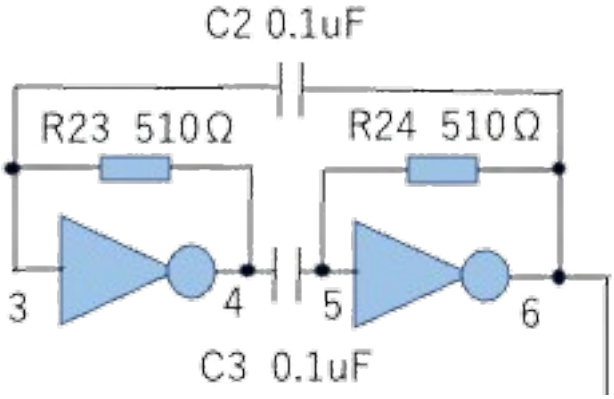
The purpose of this project, along with illustrating the general function of SAR ADCs in general, is to build and understand the clock used in the final circuit. This is key to understanding the timing and speed of the final ADC build.

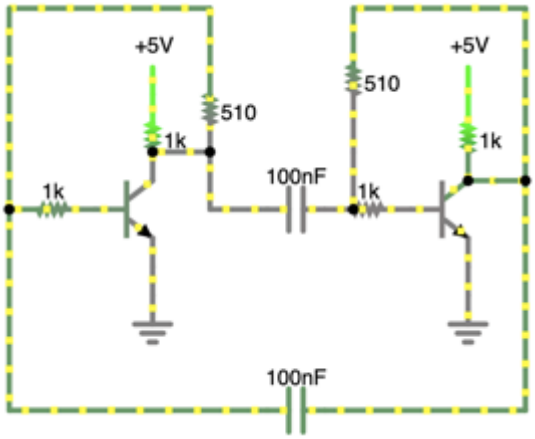
## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/SARADC/index.html#task
Yamada's Blog Post: https://hackaday.io/project/181826-homemade-successive-approximation-register-adc/details
Falstad Clock Simulation: http://tinyurl.com/yuvbsn9d

## Procedure

The circuit for the clock used in the SAR ADC is shown to the right. At this point in this report, building such a circuit is a mostly trivial task. However, to understand it is very different. This clock circuit is quite unconventional, and while it on face value may look normal there are some peculiar things going on, namely the "feedback" resistors which bridge the two sides of inverters and the fact that there are two sides of the circuit connected only by capacitors.





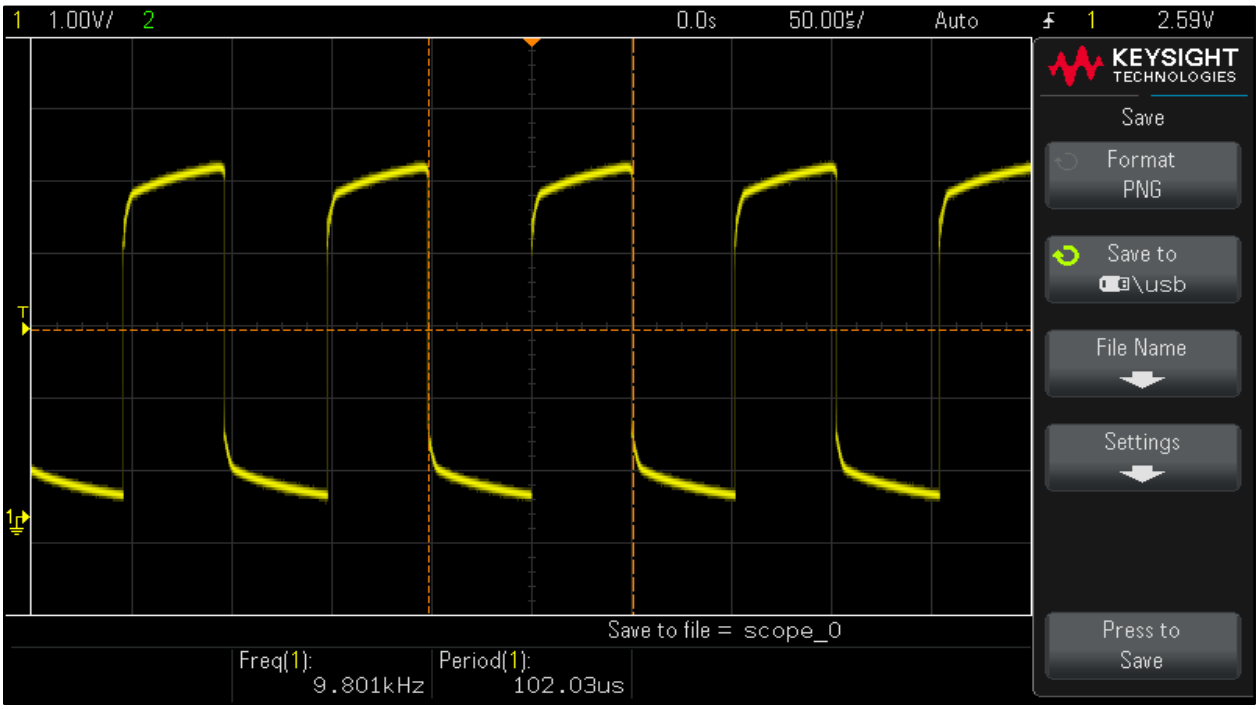

The functioning of this circuit relies on the functioning of capacitors at high frequencies, where they begin to approach an ideal wire. This idea was discussed already, and is all about the capacitive reactance of the capacitors. In this circuit, the capacitors act as wires while discharging. Capacitors resist changes in voltage, so the capacitors attempt to stabilize the voltage across them by discharging. Take the top capacitor which is currently discharging. It is allowing current to flow from the output of the right inverter (which has been recreated with a transistor) to ground through the other inverter. However, when this capacitor is sufficiently drained it will not be able to allow enough current through and the feedback resistor on the inverter to the right will build up electrons on the input of the right inverter and flip the state of the circuit where the process will repeat with the roles of each inverter switching.

Through testing it would appear that the relationship between frequency and resistance and between frequency and capacitance are roughly linearly inversely proportional, though not exactly. Note that capacitance should be measured in Farads and will thus be extremely small:

$$f \approx \frac{1}{20{,}000 \cdot R \cdot C}$$

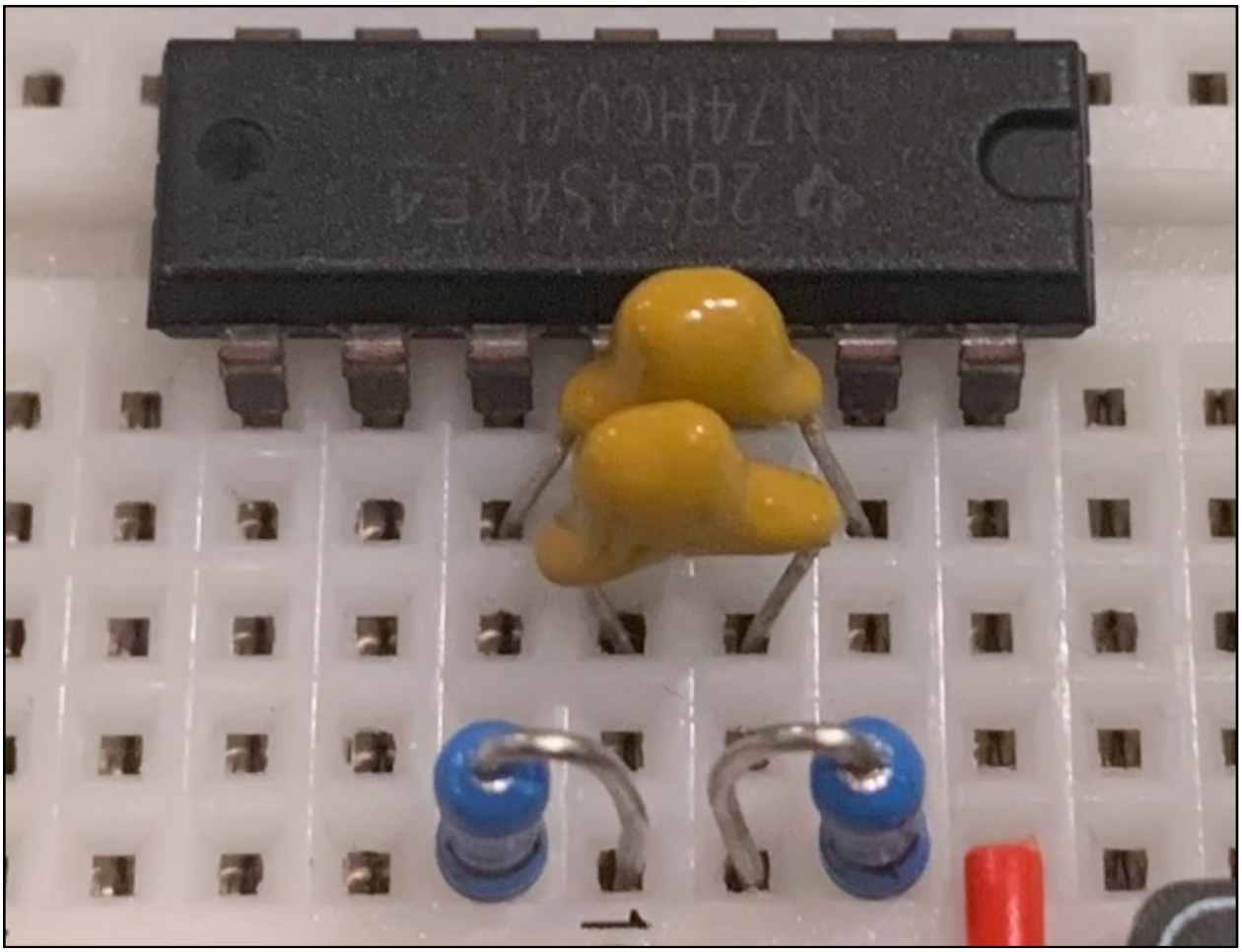
This formula works fairly well, placing the expected value for the clock build in this project at 9.9 kHz. The actual value is closer to 9.8 kHz as shown below with a scope reading.

| Parts Table | |
|---|---|
| 74HC04 Hex Inverter | 1 |
| 510 Ω Resistor | 2 |
| 0.1 µF Capacitor | 2 |



## Media

Project Video: https://youtu.be/5bj5ZRa1G14



Clock Circuit

## Reflection

This project was a good first step into the world of the SAR ADC. This project seems like it will be quite challenging, but then so did CHUMP at first and now I fully understand all things CHUMP. That is to say, I have no idea how this series of projects will go, but I am looking forward to them nonetheless.

# Project 3.2.2 SAR ADC: R/2R Ladder DAC

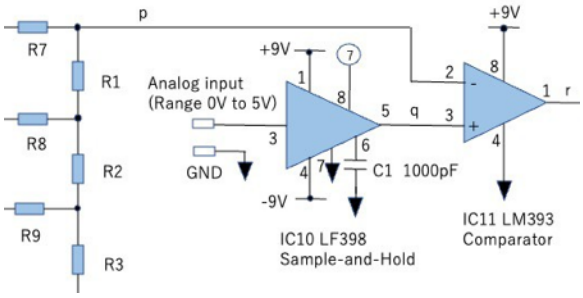
## Purpose

This project implements the DAC into the ADC in addition to the sample and hold circuit and comparator. With these components it is possible to manually simulate a conversion with the circuit, giving a functional demonstration of how the final ADC will work.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/SARADC/index.html#DAC
Falstad Simulation: https://bit.ly/49q2EXz

## Procedure

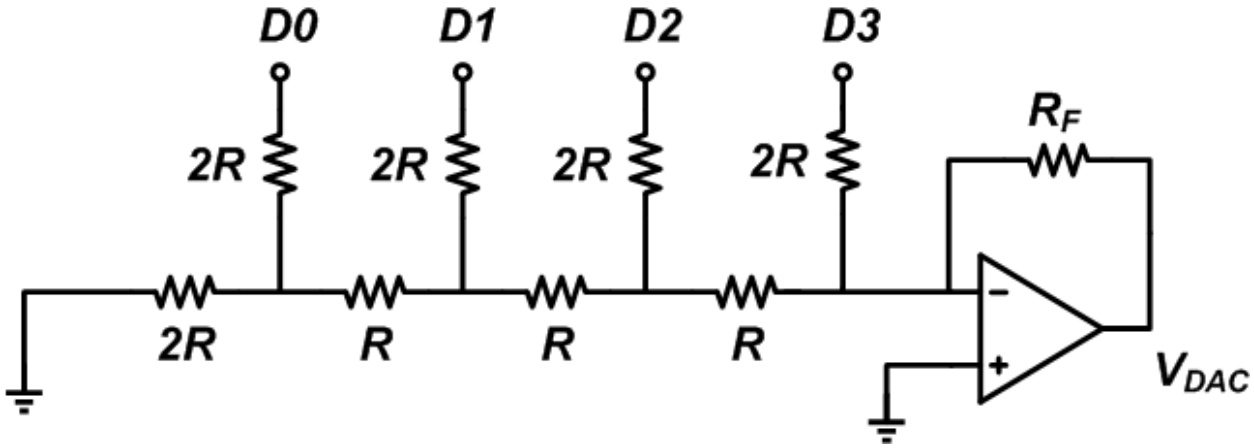



This project is broken into two distinct parts. First is the DAC within the ADC, and the second is the sample & hold and comparator portion of the circuit. The comparator is used to compare the value presented by the DAC to the sample voltage, and eventually its output will be used to set the SAR bits for subsequent comparisons in the coming clock cycles. The sample and hold IC is used to keep a steady sample voltage during the conversion, and finally part of the R/2R DAC is shown on the left side of the image (the resistor chain) to produce test voltages for the circuit to use.

## R/2R DAC

An R/2R DAC is one of the most common DAC circuits built with a resistor ladder with values of R and 2R, hence the name R/2R DAC. An R/2R DAC provides a way to directly translate binary values into analog outputs, which limits the DAC to a large yet still discrete set of analog outputs, though this normally is enough accuracy as is needed.

Below is a diagram of an R/2R DAC. It is built up of a chain of resistors which could extent as far as desired. In this diagram there is a comparator on the output of the DAC in order to maintain a stable output voltage regardless of the load, though this is not necessary in the application within this SAR ADC as the comparator which uses the DAC as an input has a high input impedence and thus almost no draw.



Before understanding how R/2R DACs work, it is important to understand that the resistance on any one bit is equivalent to 2R when looking left. This occurs through a decay chain throughout the network. To see this one must assume all bits are 0. It begins on the furthest left bit: the resistance on the first 2R resistor is 2R. However, since there are now two 2R resistors in parallel, the total resistance is just R. Now the first segment is gone, so the second bit sees an R and R in series, or simply 2R. This bit also decays to simply R, and so on and so on. This means that any one bit can look back to the left and see the total resistance in that direction as 2R if it were to turn on.

Now it is time to solve for the voltage generated by turning on any bit. Take bit 2, where 1.25 V out is expected. The resistance to the left of the bit 2 2R resistor is 2R, and to the right is 3R. These resistors are in parallel, so really the voltage present at the bottom of bit 2's 2R resistor is equal to the formula below using some simple circuit analysis:

$$V_{Bit\ 2} = V_{supply}\left(1 - \frac{2R}{2R + 6R/5}\right)$$

From there we can calculate the voltage presented to the output. To make it to the output, this previously calculated voltage must pass through R which is then connected to ground through 2R. Thus, the output voltage is the following:

$$V_{out} = V_{supply}\left(1 - \frac{2R(2R)}{3R(2R + 6R/5)}\right) = 1.25\ V$$

While it is possible to break down every single bit into equivalent resistances for resistors in parallel and in series, it is quite tedious. Luckily, the pattern between bit *n* and its output voltage is quite well documented (note that *n* begins at 0, going right to left according to the image from the previous page):

$$V_{out} = V_{supply}\left(\frac{1}{2^{n+1}}\right)$$

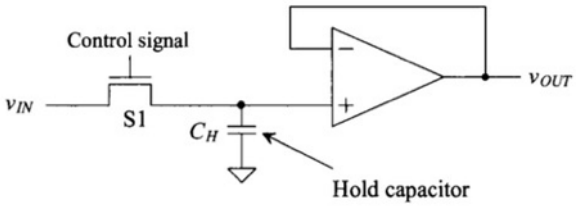
This is the idea behind the entire DAC. Each bit has a weighted output value proportional to its binary value, making an R/2R DAC perfect in this case. By simply presenting a binary value on the input bits, the analog output will be generated on the output of the DAC with no extra logic.

One thing to note while dealing with R/2R DACs is the precision of the resistors used. Each bit quickly becomes increasingly precise, and any small inaccuracies in resistor values become quite important. For example, this project uses a 7-bit DAC with resistor with an accuracy within 1% of their supposed value. This means that the last bit is useless, as its precision is 1/128 of a volt, which is more precise than the resistor itself which has a precision of 1/100. This essentially means that the least significant bit is more dependent on the resistor used than the actual voltage being read by the DAC, and technically only the top six bits of the ADC output will be significant. However, this is mostly a technicality and would only matter in actual ADC implementations where it is important that the less significant bits of the DAC have increasingly precise resistor values.

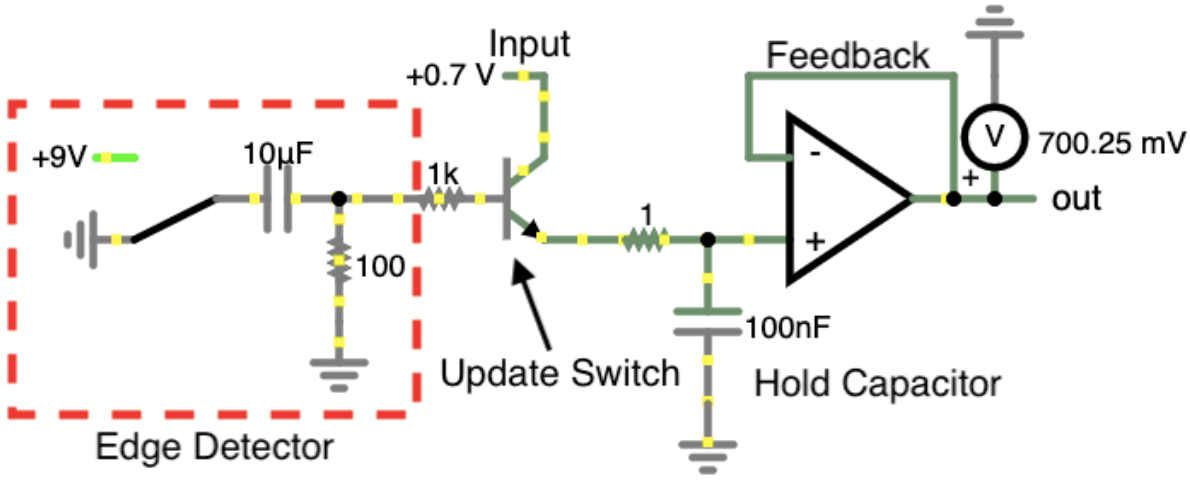
## Sample and Hold

While it is not beyond the difficulty of this report to build a sample and hold circuit from scratch it is simply not worth the space or components, and thus this circuit uses a LF398 sample and hold IC. However, it is still worth knowing how this IC functions internally. As discussed in the previous report, a sample and hold circuit is meant to maintain an analog voltage output indefinitely which is read on the rising edge of a control signal (though sometimes during the entire high period).

This stored value is kept inside a hold capacitor, as shown in the diagram to the right. This capacitor is essentially disconnected from the rest of the circuit when in the hold phase, and during the sample phase is allowed to charge to the input voltage. One important thing to note while designing a sample and hold circuit is the size of the hold capacitor. A smaller hold capacitor will have a lower sample time, but as the value is read there will naturally be an energy draw on the capacitor which will inherently alter its stored voltage, essentially making the circuit useless. This is solved with a comparator with a feedback layout as shown. In this configuration the comparator will output the voltage presented to the positive input, but now any current draw on the output (i.e. reading the voltage) will draw from the comparator and thus not affect the stored sample.




Many sample and hold circuits online are quite overcomplicated, especially those that are edge triggered. The specific sample and hold circuit below was designed for this project with the intent of being easy to understand and keeping with previously discussed ideas as the actual internal circuitry of the LF398 is less easy to understand and does not teach substantially more. To begin with, the edge detection in this circuit is quite simple. In fact, this edge detection circuit was discussed in a previous project already (The Matrix Graphing Calculator). It works by presenting an output only while the relatively small capacitor charges, which means by changing the size of the capacitor in the edge detection circuit the edge time can be changed. Next, the rising edge is passed on to the update NPN transistor, which when powered allows the input voltage to charge (or discharge) the capacitor. It is important that the edge detection circuit produces a signal long enough for the hold capacitor to completely charge/discharge. Finally, the comparator ensures a constant output sample voltage.

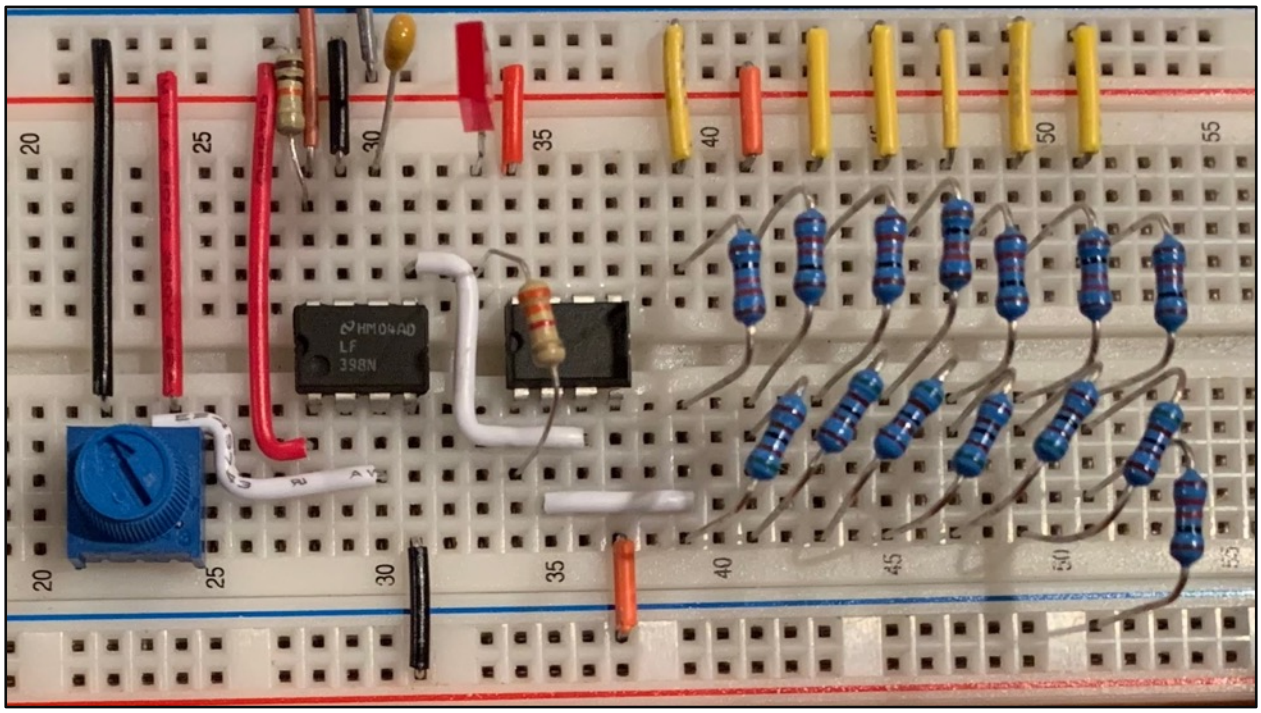


In this circuit specifically the sample and hold IC used is the LF398 as mentioned earlier. This IC is not edge triggered, which is the main difference between it and the one just developed on the previous page. The LF398 also requires 18 V to operate, or +9 V and -9 V. The hold capacitor in the LF398 is external, and while the Yamada SAR ADC calls for a 102 capacitor to be used it is more common to find 103 capacitors used in conjunction with the LF398 as they charge 10 times faster.

| Parts Table | |
|---|---|
| LF398 Sample and Hold | 1 |
| LM393 Op-Amp | 2 |
| 10.2 kΩ Resistor (2R) | 8 |
| 5.1 kΩ Resistor (R) | 6 |
| 10 nF Capacitor | 1 |
| Red LED | 1 |
| Push Button Normally Open | 1 |

## Media

Project Video: https://youtu.be/xSpVLQLIaQI




## Reflection

The thing I found most interesting about this project was the sample and hold chip. I decided to go to Falstad and attempt to create my own, and it was quite an interesting challenge. I had an idea of how it should work in my head with charging a capacitor and I built off that until I had something that worked well.

# Project 3.2.3 SAR ADC: Completed

## Purpose

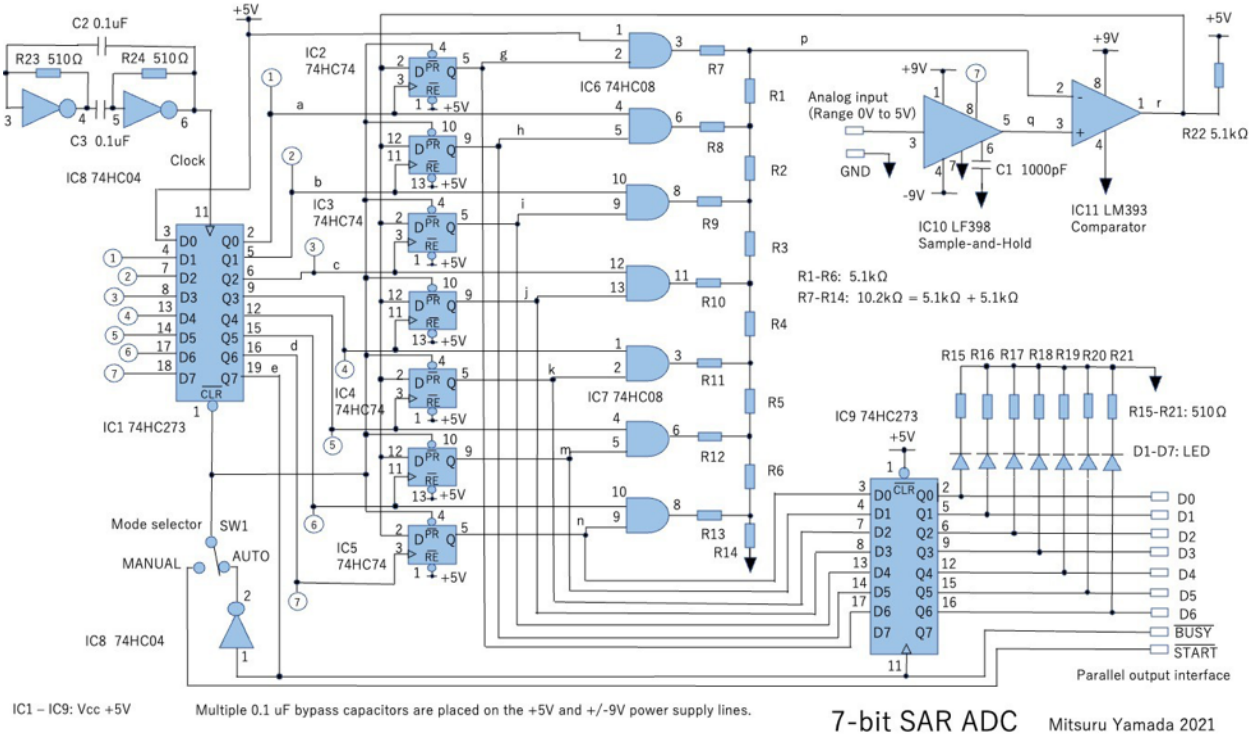
This project focuses on the final build of the SAR ADC. In this project, the SAR itself as well as the control of the DAC is implemented into the build, resulting in a fully functional 7-bit ADC.

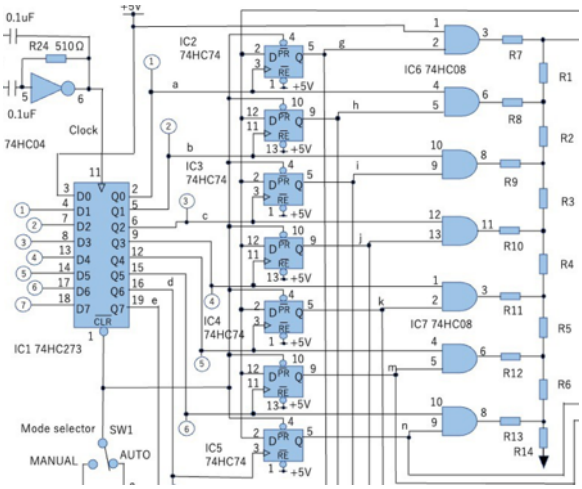
## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/SARADC/index.html#Completed
Yamada's Blog Post: https://hackaday.io/project/181826-homemade-successive-approximation-register-adc/details

## Procedure

This project begins with the first full look at the schematic of the SAR ADC. Up to this point the top right and left segments as well as the resistor ladder have been completed, leaving the rest of the circuit to be finished in this project. The rest of the circuit can be broken into three main segments. First is the SAR itself, which is built out of a chain of D-flip-flops. The SAR controls the DAC through the DAC control logic, which consists of an 8-bit register used as a shift register and a series of AND gates. Finally, another 8-bit register is used to maintain the output of the ADC during the conversion phase.

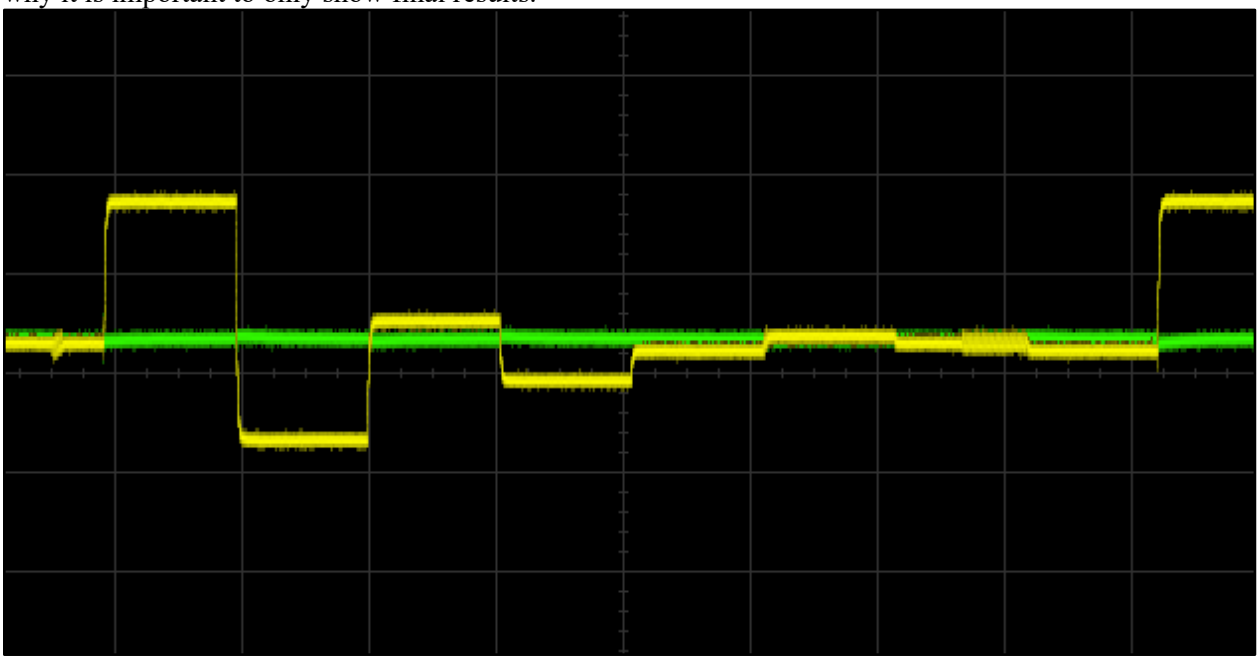





To the right is the most difficult part of the entire ADC to understand, the SAR and the DAC control logic. It is important to differentiate these two segments for what they are to better understand the function of the ADC. The SAR is quite simple: it is a 7-bit register used to store the approximation of the analog reading. It consists of four 74HC74 dual D-flip-flop ICs, and is located in the middle of the image. It is the value of this register that is shown on the output of the entire ADC at the end of each conversion. However, simply using this output as feedback for the DAC will not provide any intelligent result. Instead, a shift register is used to scrutinize one bit of the output at a time. Starting on the first cycle, all bits in the SAR are high and all outputs of the shift register are low, meaning that the only bit of the DAC that is on is the most significant bit. On the next cycle, a high signal is shifted to the clock pin of the first flip-flop, and depending on the comparison result from the comparator it either is set high or low. This signal also sets the second bit of the DAC. On the next cycle, the second flip-flop receives a rising edge on its clock pin and it updates, also setting the third bit of the DAC high. The process repeats like this, successively setting each of the seven bits in the DAC one cycle at a time.

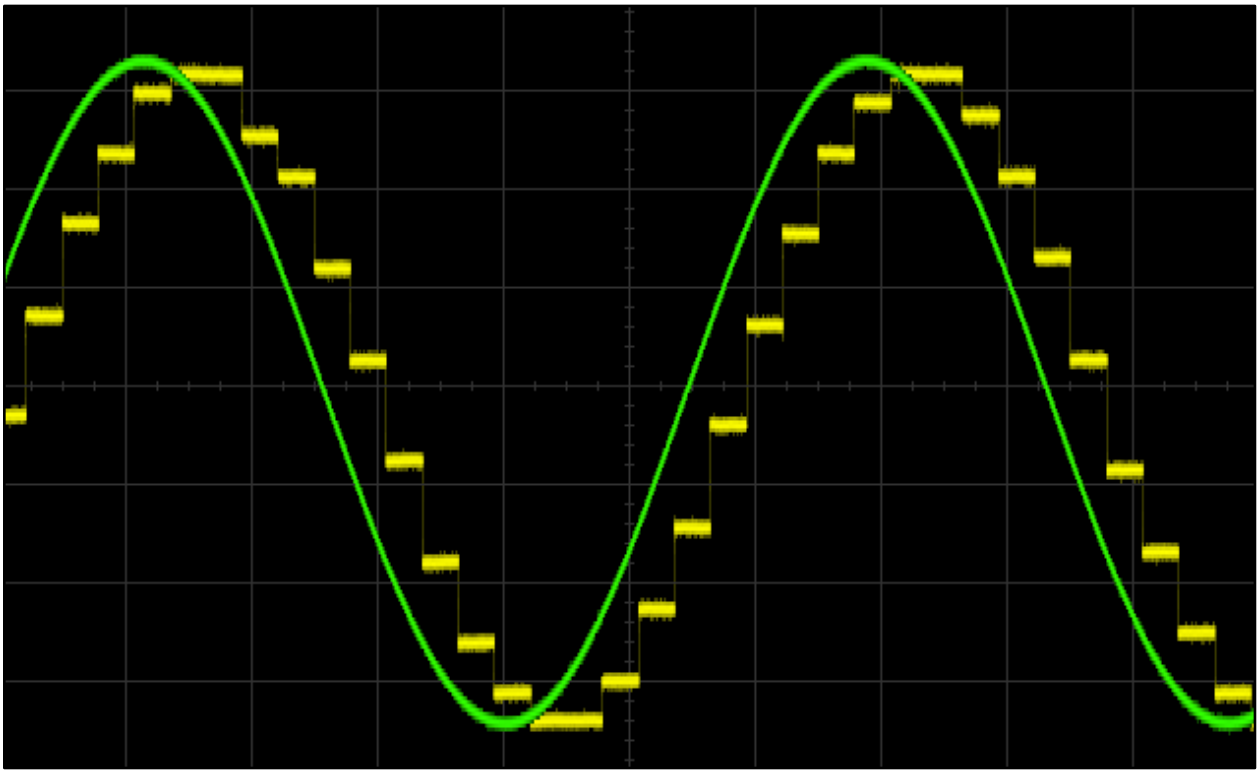
On the eighth tick of the clock, the shift register is cleared and all outputs are set low. In addition, all of the bits of the SAR are set high. On the tick before this, the sample and hold circuit takes a new sample voltage. These actions culminate to the beginning of a new approximation cycle of a new sample voltage, and thus the process repeats indefinitely.

As the current value in the SAR is constantly changing throughout the approximation cycle, a register is used to store the output of the DAC and will update at the end of each approximation, meaning that the output will only ever display the final result of the conversion and none of the intermediate steps required to obtain the final result. Below is an image of what the output looks like during the approximation, and why it is important to only show final results.



ADC on the Oscilloscope

In order to see the accuracy of the ADC another DAC can be used such that the output of the ADC is fed into the DAC, essentially showing the analog version of the digital conversion. Below is an image from the oscilloscope of the ADC approximating a sine wave. The clear execution time of the circuit is evident here as well (the reason for the phase shift):

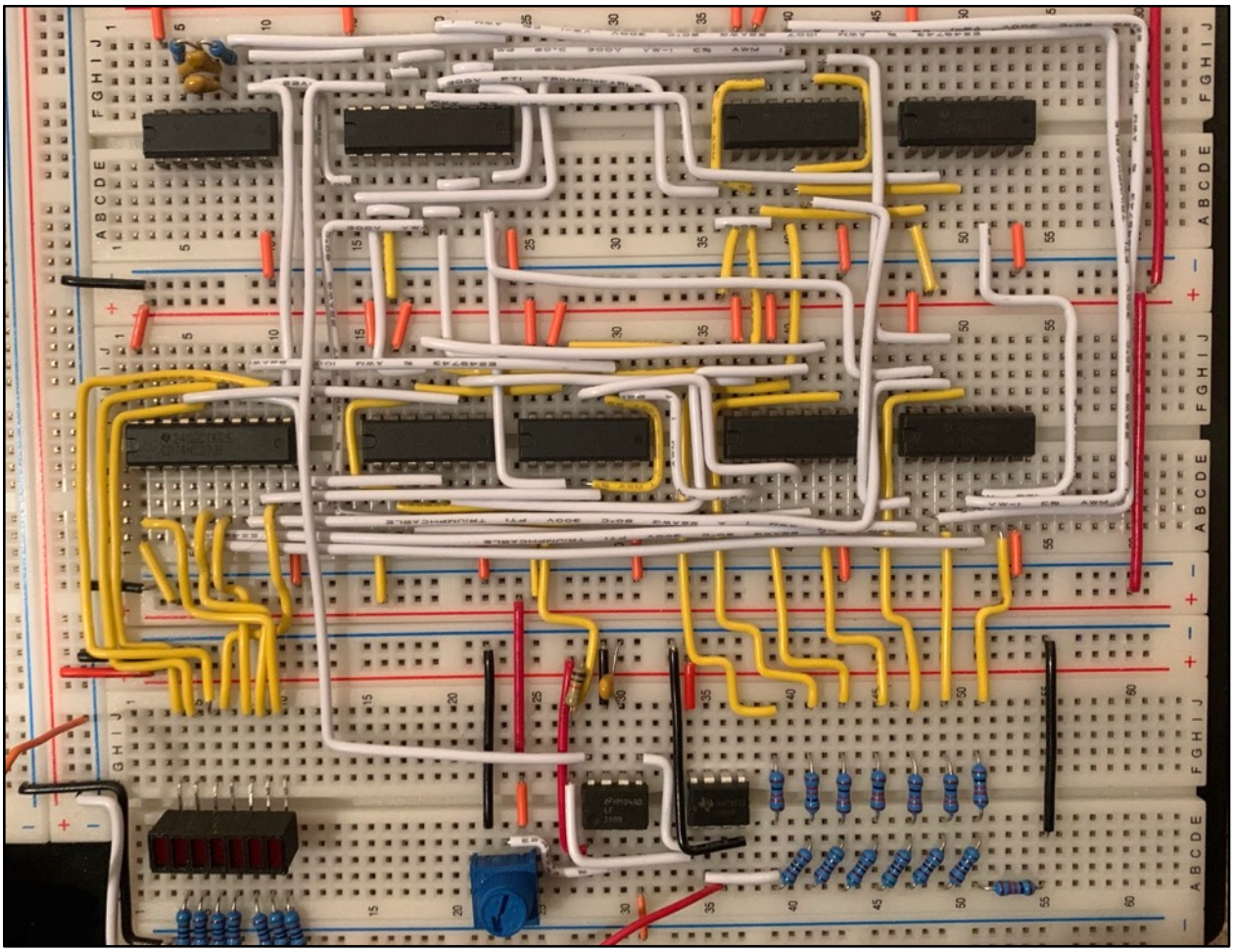


The final thing to discuss about the ADC is how one is meant to interpret its output. The output comes in the form of a 7-bit number, and this number can be thought of as the ratio of the input voltage to the range of the source voltage, with this ratio being the 7-bit number divided by 127 (the maximum possible output). This means that to determine the input voltage one must simply divide the output number by 127 and then multiply by the input range, in this case 5 V. So, for example if the 7-bit number is 55, the voltage being read is $55 \div 127 \cdot 5\,V = 2.17\,V$. Using a DMM to confirm, this value is accurate up to two decimal points, which is quite precise.

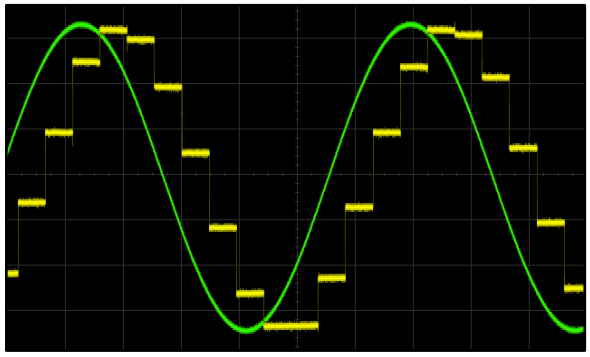
| Parts Table | |
|---|---|
| 74HC273 Octal D-Flip-Flop | 2 |
| 74HC74 Dual D-Flip-Flop | 4 |
| 74HC04 Hex Inverter | 1 |
| 74HC08 Quad AND Gate | 2 |
| LM7805 Voltage Regulator | 1 |
| LF398 Sample and Hold | 1 |
| LM393 Op-Amp | 1 |
| 10.2 kΩ Resistor (2R) | 7 |
| 5.1 kΩ Resistor (R) | 16 |
| 10 nF Capacitor | 3 |
| Red LED | 7 |
| 10 µF Polarized Capacitor | 2 |
| 9 V DC Power Supply | 2 |

## Media

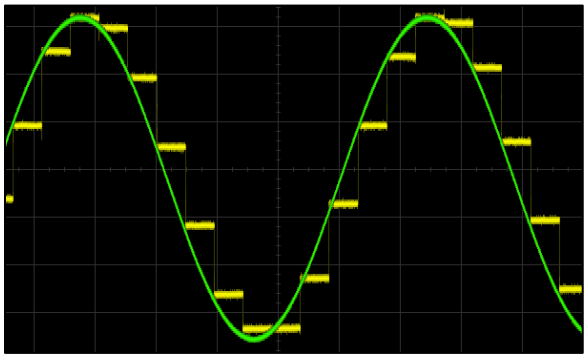
Project Video: https://youtu.be/bu2GMxEixDI



Final ADC Build



Waveform Approximation



Waveform Approximation (Phase-shift Corrected)

## Reflection

I found the write up of this project to be somewhat challenging. It felt like there was not much left to say about the ADC after covering it thoroughly in the previous reports. This largely reflects how my understanding also did not change too much, I spent the time early on to learn and understand the ADC before I built it, so this project was really all about simply wiring it up and watching it work. I am not sure which style I prefer: this route, or the standard learn as you wire/write. I will say that this was a much more relaxed wiring experience compared to CHUMP as I already understood exactly how everything would work together in the final build from the moment I placed the first wire.

The project itself was very interesting. The design of this ADC is very smart and minimal and it has a lot to teach about designing efficient circuits for specific purposes. I am sure that if I had designed my own schematic from the same idea/algorithm I would have ended with many more ICs, especially for the control of the DAC.

In the end, it was very rewarding to see the final build come to life when I plugged it in and it started spitting out conversions. I had this same feeling when I brought it to the scope and saw the approximation in action and the result following a sine wave, which I think produced some quite compelling images.

# Project 3.20c Medium ISP: Robot Arm

## Theory

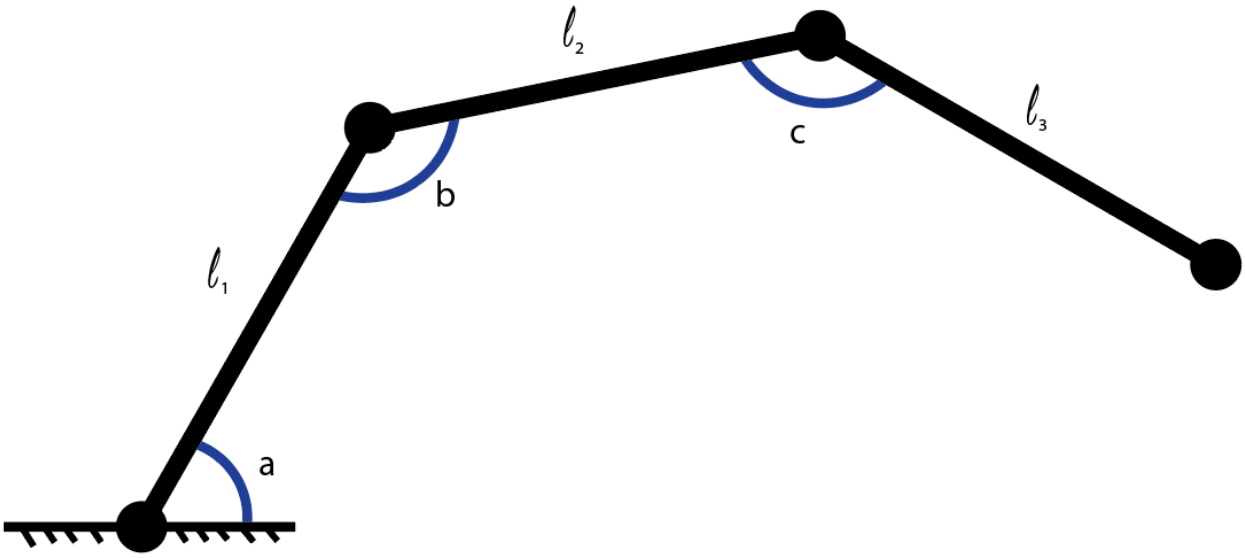
This ISP explores the creation of a robot arm from servos and 3D printed parts, powered by an Arduino Nano. In doing so, this project teaches about key principles of mathematics, algorithms, and design. Those three domains are the sub-topics of this project, and each topic poses a problem. In order to make a robot arm one must be able to mathematically model the arm, must be able to algorithmically solve the mathematical problem posed by the arm, and must be designed in such a way that the other two topics are allowed to shine. The first two topics of this project are so broad that they constitute their own field of study: inverse kinematics (IK). Inverse kinematics, or the inverse kinematics problem, is an algorithm or set of equations that gives a set of output conditions to move a linkage of joints to a specific set of coordinates in 3D space, and is integral to this project.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/2223/ISPs.html#logs
FABRIK Paper: http://andreasaristidou.com/publications/papers/FABRIK.pdf

## Procedure

The very first step to designing a robot arm is to model it mathematically. It is much easier to begin the design process with knowledge of how the final product will function. This project began with a basic sketch of the arm with labeled points and angles. Below is a recreation of that initial sketch. It is important to be familiar with the angles and names used in this sketch as they will appear throughout the mathematical process of modelling the arm and making it move.




Note that the diagram above shows a two-dimensional view of the arm. There will be an additional pivot point around the z axis which can be seen from the top or bottom. However, this is not necessary for the modelling of the arm for reasons that will be discussed later.

## Mathematical Model

With a very basic design for arm complete, the next step is to model the coordinates of the end point (normally called end effector or EF) depending on the angles of each joint. This is the opposite of IK, and is called forward kinematics. This model is important both for solving the IK and for checking angle outputs when using other algorithms to solve the IK. The easiest way to do this is to use a three-part sum for each the x and y coordinate and calculate the horizontal and vertical component of each segment individually. The first segment is simple:

$$x_a = \ell_1 \cdot \cos(a)\,, y_a = \ell_1 \cdot \sin(a)$$

For the second segment the derivation is a bit trickier as the angle between the segment and the ground is a combination of both angles $a$ and $b$. However, with a bit of angle theorem the second formula is also rather straight forward:

$$x_b = \ell_2 \cdot \cos(a + b - \pi)\,, y_b = \ell_2 \cdot \sin(a + b - \pi)$$

For the third and final segment the angle becomes quite a bit harder to find until considering that it is the exact same scenario as the second segment except $a$ is now $a + b - \pi$ and $b$ is now $c$. Considering this, the equation for finding the components of the last segment are the following:

$$x_c = \ell_3 \cdot \cos(a + b + c - 2\pi)\,, y_c = \ell_3 \cdot \sin(a + b + c - 2\pi)$$

Once again, the total coordinates of the EF will simply be a sum of all three segments. Using these segment equations, the equations for the components of the EF are:

$$\begin{aligned} x_{EF} &= \ell_1 \cos(a) + \ell_2 \cos(a + b - \pi) + \ell_3 \cos(a + b + c - 2\pi) \\ y_{EF} &= \ell_1 \sin(a) + \ell_2 \sin(a + b - \pi) + \ell_3 \sin(a + b + c - 2\pi) \end{aligned}$$

While this set of equations is correct, it is very far from its most simple form. The second and third terms can be cleaned up quite a bit. The first thing to notice is that adding or subtracting any multiple of $2\pi$ from any sinusoid with a period of $2\pi$ has no effect. Thus, the equation can be simplified by removing the redundant $2\pi$. Next, the middle term needs some work. Starting with the $x_{EF}$, here is the expanded formula for $\cos(\theta_1 + \theta_2 + \theta_3)$:

$$\begin{aligned} &\cos(\theta_1 + \theta_2 + \theta_3) = \cos(\theta_1 + \theta_2)\cos(\theta_3) - \sin(\theta_1 + \theta_2)\sin(\theta_3) \\ &= \cos(\theta_1)\cos(\theta_2)\cos(\theta_3) - \sin(\theta_1)\sin(\theta_2)\cos(\theta_3) - \cos(\theta_1)\cos(\theta_2)\sin(\theta_3) + \sin(\theta_1)\sin(\theta_2)\sin(\theta_3) \end{aligned}$$

This may look more complicated, but with this we can substitute $-\pi$ for $\theta_3$. This ensures that all instances of $\sin(\theta_3)$ will be 0 and $\cos(\theta_3)$ will be -1. By taking this into account, the formula becomes:

$$\cos(\theta_1 + \theta_2 - \pi) = -(\cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2)) = -\cos(\theta_1 + \theta_2)$$

Similarly, $\sin(\theta_1 + \theta_2 + \theta_3)$ is expandable:

$$\begin{aligned} &\sin(\theta_1 + \theta_2 + \theta_3) = \sin(\theta_1 + \theta_2)\cos(\theta_3) + \cos(\theta_1 + \theta_2)\sin(\theta_3) \\ &= \sin(\theta_1)\cos(\theta_2)\cos(\theta_3) - \sin(\theta_1)\sin(\theta_2)\sin(\theta_3) + \cos(\theta_1)\sin(\theta_2)\cos(\theta_3) + \cos(\theta_1)\cos(\theta_2)\sin(\theta_3) \end{aligned}$$

Once again, the fact that $-\pi$ can be substituted for $\theta_3$ will help simplify the equation. As a reminder, this ensures that all instances of $\sin(\theta_3)$ will be 0 and $\cos(\theta_3)$ will be -1. Here is the simplified expression:

$$\sin(\theta_1 + \theta_2 + \theta_3) = -(\sin(\theta_1)\cos(\theta_2) + \cos(\theta_1)\sin(\theta_2)) = -\sin(\theta_1 + \theta_2)$$

Using these simplified expressions, the final equations for the coordinates of the EF can be written down:

$$x_{EF} = \ell_1 \cos(a) - \ell_2 \cos(a+b) + \ell_3 \cos(a+b+c)$$
$$y_{EF} = \ell_1 \sin(a) - \ell_2 \sin(a+b) + \ell_3 \sin(a+b+c)$$

These expressions can be used to check what the final coordinates will be with a given set of angles, but cannot produce a set of angles from given coordinates. There is still use to them for checking results, but in order to solve angles explicitly, there needs to be as many equations as variables. What is possible is to constrain one angle with some physical meaning, and then express it in terms of the other angles. A good choice is to choose for the second segment of the arm to be parallel to the imaginary line between the base of the arm and the target coordinates at all times. This allows for all positions to still be reached while also removing the variable of $b$ from the equations. This means that $b$ will always be the following:

$$b = \arctan\left(\frac{y_{EF}}{x_{EF}}\right) + \pi - a$$

The equations can now be rewritten with this new $b$:

$$x_{EF} = \ell_1 \cos(a) - \ell_2 \cos\left(\arctan\left(\frac{y_{EF}}{x_{EF}}\right)\right) - \ell_3 \cos\left(\arctan\left(\frac{y_{EF}}{x_{EF}}\right) + c\right)$$
$$y_{EF} = \ell_1 \sin(a) - \ell_2 \sin\left(\arctan\left(\frac{y_{EF}}{x_{EF}}\right)\right) - \ell_3 \sin\left(\arctan\left(\frac{y_{EF}}{x_{EF}}\right) + c\right)$$

Either of these new equations can be used to isolate for $a$:

$$a = \arccos\left(\frac{\ell_2 \cos\left(\arctan\left(\frac{y_{EF}}{x_{EF}}\right)\right) + \ell_3 \cos\left(\arctan\left(\frac{y_{EF}}{x_{EF}}\right) + c\right) + x_{EF}}{\ell_1}\right)$$

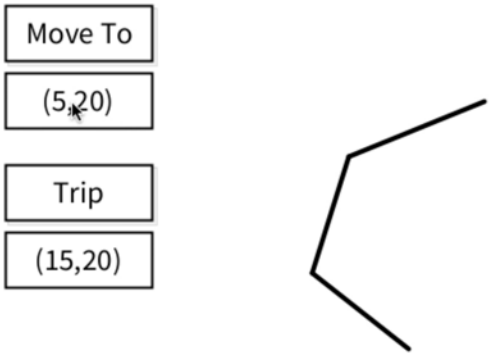
This can now be plugged in to the other equation to get an equation where the only unknown is $c$. To make things more readable we will substitute $\arctan\left(\frac{y_{EF}}{x_{EF}}\right)$ with simply $\theta$:

$$y_{EF} = \ell_1 \sin\left(\arccos\left(\frac{x_{EF} - \ell_1 \cos(\theta) + \ell_3 \cos(\theta + c)}{\ell_1}\right)\right) + \ell_2 \sin(\theta) - \ell_3 \sin(\theta + c)$$

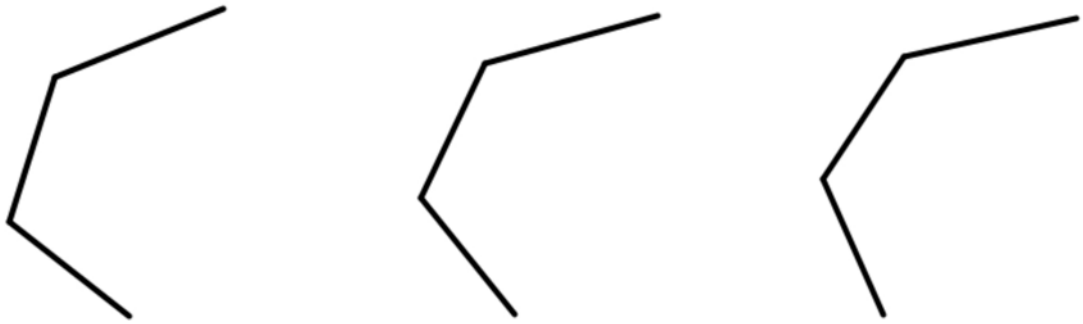
If this equation could somehow be isolated for $c$, it would mean that $a$ and $b$ could also be found. While it could likely be done by some very smart people, this report is not the place. However, even without an explicit solution, this equation is still incredibly powerful. Software makes precise approximations of $c$ and thus all three angles very easy, though not necessarily fast. Using this, accurate IK angles can be computed for a given angle using a purely mathematical approach. This is faster than using the original set of equations, which would need millions of computation cycles to attempt every set of integer degree joint angles. With this method, only 360 cycles are needed for the same precision, though admittedly each cycle takes much longer (though not 100,000 times longer, meaning this method is still faster).

## Software Simulation

Using the formula from the previous page a simulation for the arm was made using Processing. To the right is a screenshot of the sketch at a single angle. In addition to computing single angles, this software is also able to compute paths between two points using slope-based interpolation. This interpolation algorithm populates a path array with waypoints for the arm to visit in between two points, and by moving small increments between them it makes the arm appear as though it is moving linearly.



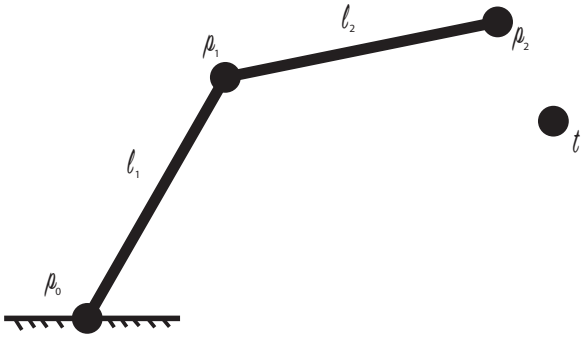

Below is a series of three images from the Processing sketch. These images illustrate how the algorithm produces smooth motion in between points. In addition to the graphical output, the sketch also prints the values of all angles to the terminal. The sketch computes a new angle, $p$, in addition to the three previously discussed angles. This is the pivot angle, which is the bird's eye view rotation of the arm. This reconciles the issue with computing a 3D problem using a 2D model. The pivot angle is determined first, and then the Pythagorean theorem is used to determine the 2D model's horizontal value, or what has been referred to as $x_{EF}$.



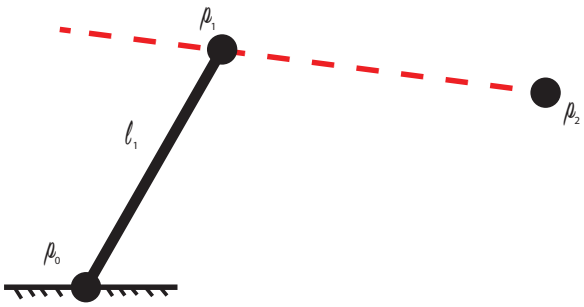
## FABRIK

While this method of approximation is relatively fast and accurate compared to the original set of formulas with three variables, it still takes tens of thousands of individual computations to arrive at a final result. While an explicit formula for each angle is not feasible, there are other faster IK solutions. One such solution is FABRIK, which stands for forwards and backwards reaching inverse kinematics. This is a vector-based method of IK. Due to this method being vector based, it works in all dimensions above one-dimensional space.

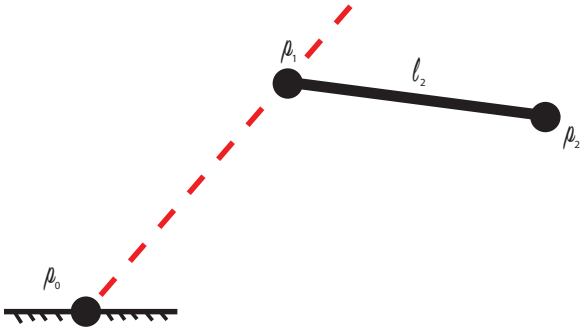
The algorithm begins with a basic linkage setup as shown to the right. In this scenario the coordinates (or vectors) of each point are known, which means that by extension so is their length with a bit of calculation. Besides the linkage itself, there must also be a target point. This is the starting condition of the algorithm; nothing happens in this step of the execution.



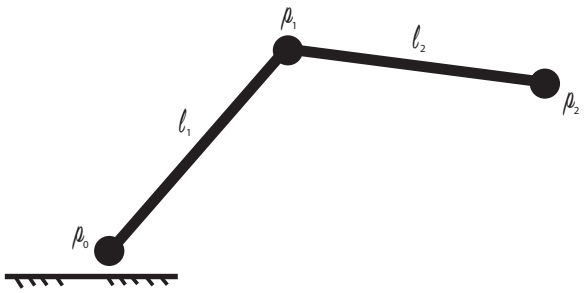
Once the algorithm has been given the information above, it moves on to the beginning of the process. The algorithm begins by moving the final point of the linkage to the target destination. Next, a line is drawn from the new position of the final point to the point before it. It is also important to note the original length in-between each point before this step, or else the next step is not possible.




In the next step two things happen. First, the second last point in the linkage is moved along the line drawn in the previous step a length of the original $\ell_1$ from the final point. Next, a new line is drawn from this new point to the point before it. Once again, it is important to store the original length in-between these points before moving the second last point in order for the algorithm to work.




Finally, the first point in the linkage is moved a length of $\ell_2$ along the new imaginary line from the middle point. This completes the forwards reaching part of FABRIK. However, the problem here is that the new location of $p_0$ no longer at the origin. This poses a problem for obvious reasons. To reconcile with this issue, the algorithm is repeated again, however this time the order of points is flipped and the target point is changed to the origin. This means that by the end of a whole cycle of FABRIK $p_0$ will remain at the origin, though all other points will likely move. This algorithm can be used with *n* points and segments, which makes it a powerful general tool for solving IK problems. While there is no rigorous mathematical proof that shows that FABRIK converges on the target point, it has been proven experimentally with studies. In fact, with only five iterations FABRIK will already be far more precise than the previous approximation algorithms.




### Recovering Angles from FABRIK

Simply knowing the coordinates of each joint in the arm is not enough to move the arm to any position. This requires the angles to move each joint to. However, the coordinates of each joint give the required information to solve for each angle. Beginning with angle $a$, the formula is very simple:

$$a = arctan\left(\frac{y_{p1}}{x_{p1}}\right)$$

The other two angles require the use of cosine law. Below is cosine law when solving for an angle:

$$c = arcos\left(\frac{C^2 - A^2 - B^2}{-2AB}\right)$$

The only problem with cosine law in this situation is that the length of *C* is not given. This means that the Pythagorean theorem will be necessary to solve for it. *C* can be expressed by the length of the addition of two vectors. The full formula for both $b$ and $c$ is shown below:

$$b = arcos\left(\frac{(x_{p_2} - x_{p_0})^2 + (y_{p_2} - y_{p_0})^2 - {\ell_1}^2 - {\ell_2}^2}{-2\ell_1\ell_2}\right)$$
$$c = arcos\left(\frac{(x_{p_3} - x_{p_1})^2 + (y_{p_3} - y_{p_1})^2 - {\ell_2}^2 - {\ell_3}^2}{-2\ell_2\ell_3}\right)$$

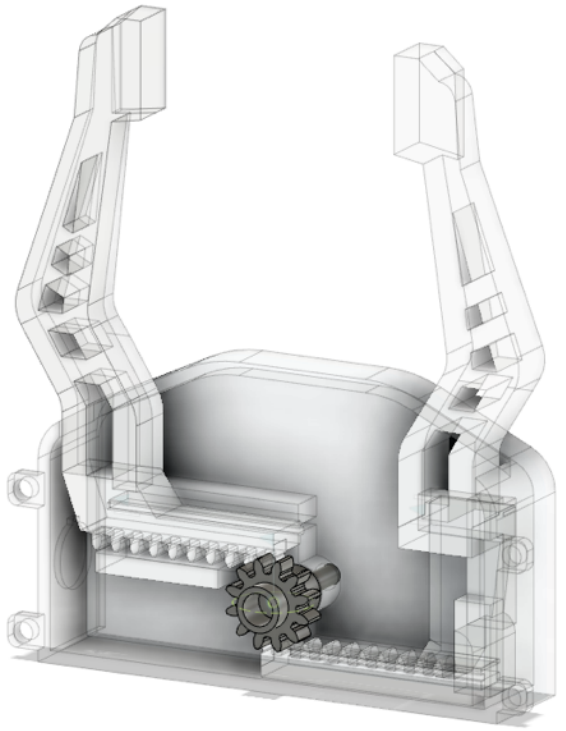
In addition to angles, FABRIK poses another problem. Since FABRIK is used in 3D space, the previous slope-based interpolation method is not possible. To solve this, a vector-based interpolation method can be applied. To do this, a vector is calculated from the initial position of a trip to the end position. From there, the direction of the vector is found by calculating the unit vector. By multiplying the unit vector with a scaler, the interpolation algorithm finds many steps along that vector between the two points. By adding these multiples of $\hat{v}$ to the vector of the first point, the algorithm populates a path with many steps along a line in-between the start and end of the trip.

## Design

The robot arm itself is designed from almost entirely 3D printed parts with servo motors as joints. The design prioritizes mechanical simplicity at most points. The only moving parts are the joints and the gripper on the end of the arm. Since everything is relatively straightforward, there is not too much to discuss about the design aspect of the arm, with the notable exception of the gripper.

| Parts Table | |
|---|---|
| Servo Motor | 4 |
| Custom 3D Printed Parts | ** |
| Arduino Nano | 1 |
| 9 g Servo Motor | 1 |
| Press Board | 1 |

To the right is an image of the gripper at the end of the arm. The design of the gripper is centered around a single gear controlled by a small externally mounted servo motor. When this gear spins, the "fingers" on either side of the gripper open or close. This linkage between the gears and the fingers is called a rack and pinion. A rack and pinion is commonly used to translate angular motion to linear motion or vice versa. There is no built-in rack and pinion tool in Fusion 360; they have to be custom built using some tricks with the standardized gear library. This design is simpler than other methods of translating angular motion to linear motion such as pistons, and the entire gripper was custom designed and built for this arm (as was every 3D printed part).



This gripper is admittedly relatively weak, but it does a good enough job of illustrating the function of the gear design and how the gripper could work well with a powerful servo. Nonetheless, the gripper can pick up some small objects.

## Media

Project Video: https://youtu.be/txEHSlE1_hg
Angle Derivation Video: https://youtu.be/d3FBmTHZoGY
Fusion 360 Files (and Code): https://github.com/Liam-McCartney/Hardware/tree/main/RoboArm

## Reflection

I really enjoyed this ISP. I grew my skillset in Fusion 360, applied some math concepts from earlier this year that I never thought I would, and learned some cool algorithms. I think that in this project I did a good job of creating a real body of work. I made mathematical proofs, simulations, animations, and the final product. I have chosen to leave most things on the GitHub page so I can look back later.

## Code

This project contains many pieces of software and code. These include the visual simulation model in Processing, the sketch running on the Arduino Nano itself, and a pure C++ file and executable for computing the FABRIK model. There are over 1000 lines of combined code for this project, and thus it is unreasonable to put it all in this document. All the full code can be found at the GitHub page for this project: https://github.com/Liam-McCartney/Hardware/tree/main/RoboArm. Nonetheless, here is perhaps the most important section of code. Below is a single iteration of the FABRIK algorithm:

```
void iterate(vect target) {
  //Completes a single iteration of FABRIK
  float reach = l1 + l2 + l3;
  if (reach < vectLength(target)) {
    Serial.println("Not Possible! Coordinate is too far!");
  } else {
    p3 = target;
    //Set p3 to the target
    p2 = vectSum(scalar(unitVect(vectorSub(p2, p3)), l3), p3);
    //Set p2 to l3 away from p3 on the line from p3 to p2
    p1 = vectSum(scalar(unitVect(vectorSub(p1, p2)), l2), p2);
    //Set p1 to l2 away from p2 on the line from p2 to p1
    p1 = vectSum(scalar(unitVect(vectorSub(p1, p0)), l1), p0);
    //Set p1 to l1 away from p0 on the line from p0 to p1
    p2 = vectSum(scalar(unitVect(vectorSub(p2, p1)), l2), p1);
    //Set p2 to l2 away from p1 on the line from p1 to p2
    p3 = vectSum(scalar(unitVect(vectorSub(p3, p2)), l3), p2);
    //Set p3 to l3 away from p2 on the line from p2 to p3
  }
}
```

# Project 3.30c Long ISP: Rocket Telemetry

## Purpose

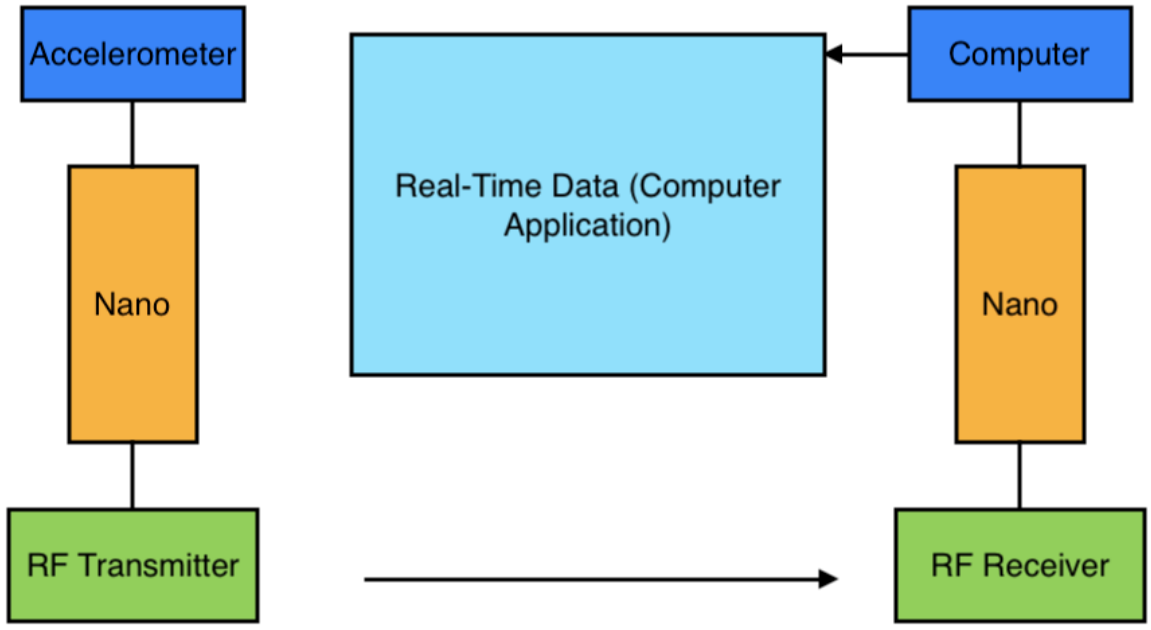
This project expands upon software and communications concepts in order to create a working application for rocket telemetry. The computer-side application teaches about user interface concepts while the hardware build poses the challenge of fitting a design into a small space.

## References

Project Description: http://darcy.rsgc.on.ca/ACES/TEI4M/2324/ISPs.html

## Procedure

This project is best broken up into both a hardware and software section, but before that it is good to discuss the general idea of this project and the flow it follows. Rocket telemetry means gathering data from a rocket and sending it back to some device still on the ground while in flight. In this project, a three-axis accelerometer is used to gather roll, pitch, and acceleration data about a rocket and is sent back to a computer on the ground running a real-time monitoring application which graphs acceleration, velocity, and altitude over time and shows a live tilt view of the rocket. Below is a diagram that shows the flow-diagram for the entire syste



## Hardware

The hardware in this project mainly consists of the accelerometer and RF transceiver components. The accelerometer used for this project is the ADXL345, which is a three-axis I2C and SPI accelerometer. In this project it was used through I2C as it is easier to do with the use of the internal Wire library in the Arduino IDE.


Most accelerometers (including this one) do not measure actual acceleration. That is to say, they do not measure a velocity change over time. Instead, they measure the acceleration in a scale of $g$, or 9.81 $^{m}/_{s^2}$, except for on the z-axis. On this axis, the reading at rest while the accelerometer is lying flat will be 1, meaning or 9.81 $^{m}/_{s^2}$. This is done by the manufactures on purpose as most people simply do not care if their acceleration is in g-scale or not, but it does mean that for this application there will be some necessary conversions.

To convert from this strange method of measuring acceleration, the following equation can be used:

$$|\vec{a}_{real}| = g \cdot \left( \sqrt{\vec{a}_x{}^2 + \vec{a}_y{}^2 + \vec{a}_z{}^2} - 1 \right)$$

This equation gives the physics-defined acceleration, which is necessary for the computer to calculate the velocity and altitude of the rocket at a later point.

Accelerometers themselves are interesting devices as they apply physics concepts to electronics. There is not circuit or combination of logic gates that will be able to measure the acceleration of a system. To accomplish this requires the implementation of physics concepts to electronics. There are two methods for doing this. One relies on the piezoelectric effect, and one relies on capacitance. In the piezoelectric effect method, one small quartz crystal is used for each axis, and when the accelerometer accelerates the crystals slightly deform. This deformation creates a voltage proportional to the force on the crystal, which physics tells us is proportional to the acceleration. Similarly, capacitance-based accelerometers like the ADXL345 use small spring-suspended masses, and when acceleration causes the masses to move a difference in capacitance is measured between the mass and its casing. This capacitance change is once again proportional to the acceleration experiences by the accelerometer.

The final necessary thing to discuss about the accelerometer is the range. The ADXL345 has four different acceleration it can use. This range can be changes in software using the I2C protocol, and changes the maximum measure acceleration. Since model rockets can easily exceed 2 g of acceleration it is necessary to change this value in the software from the present value. However, this does mean that the accuracy of the data will be less than with the 2 g scale.

***Register 0x31—DATA_FORMAT (Read/Write)***

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| SELF_TEST | SPI | INT_INVERT | 0 | FULL_RES | Justify | Range | |

**Table 21. *g* Range Setting**

| Setting | | |
|---|---|---|
| **D1** | **D0** | ***g* Range** |
| 0 | 0 | ±2 *g* |
| 0 | 1 | ±4 *g* |
| 1 | 0 | ±8 *g* |
| 1 | 1 | ±16 *g* |

The RF transceiver used in this project is a clone of the popular HC-12. This device has a range of up to 1.8 km in open air, which is important for a model rocket project as at the highest the rocket will be over 200 m above from the computer, and can drift on its decent. Unfortunately, there is very little selection of transceivers in between 100 m and around 1.5 km, so this very long-range device is the best there is for the relatively low price. The HC-12 comes with an antenna shown on the right, but with this included antenna the range is actually less than 1.8 km and makes this the perfect all round transceiver wich does not require an antenna to be purchased.

The HC-12 is an SPI device. It has an onboard STM microcontroller used to encode and deconde transmissions, which makes communication with it quite simple. All the user has to do is write anything to the transmitter on the SPI bus, and it will then broadcast it to the receiver. From there, the receiver will send the received data to the computer-side Nano through SPI. Technically the transceivers could be directly connected to the SPI pins of the Nanos, but when using the Serial Monitor this causes to many problems and thus the SPI communication with the devices is emulated with the software serial library, and communication is limited to 9600 baud.

## Software

The software side of this project includes three separate pieces of code. One is for the transmitter Nano, one for the receiver Nano, and a final one for the application built in processing. Starting with the receiver side (since it is the most simple), all this sketch has to do is receive transmitted data and print it again over SPI to the computer.

The transmitter side is a bit more complicated. The transmitter has to configure the ADXL345 as well as read from it and then send that over the air to the receiver. The ADXL345 must be calibrated every time the sketch is run. This is done by writing correction information to the correct registers using the wire library in the Arduino IDE. According to the datasheet for the ADXL345, registers 1E to 20 are offset calibration register, the contents of which are added to the readings for each axis. These values are in two's compliment, so values may also be subtracted from.

In addition, as mentioned earlier the range of the accelerometer must be set to 16 g in the DATA_FORMAT register. This is done by reading from the register, clearing the last two bits through a bit mask, and then using a bitwise-OR operation to set the last two bits to 1. Technically it is only required to use a bitwise-OR to set the range to 16 g, but for other ranges this might not work if one of the bits were already set high.

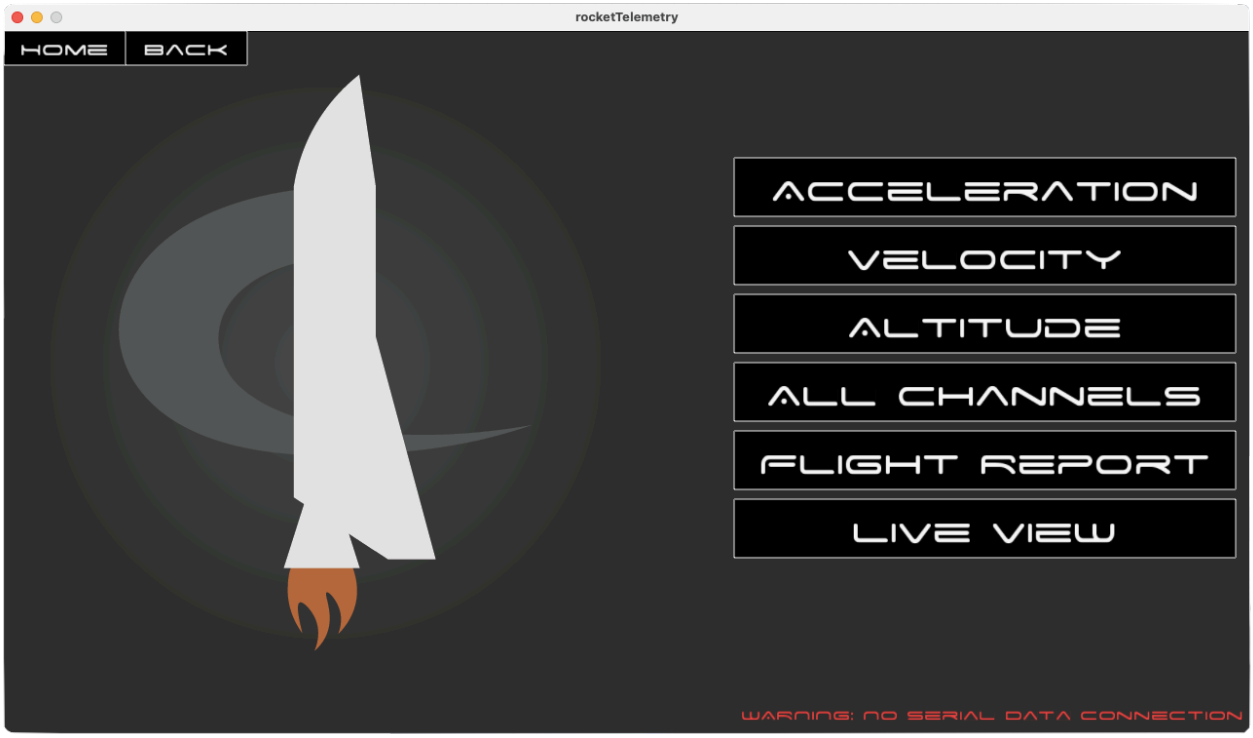
***Register 0x31—DATA_FORMAT (Read/Write)***

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| SELF_TEST | SPI | INT_INVERT | 0 | FULL_RES | Justify | Range | |

**Table 21. *g* Range Setting**

| Setting | | |
|---|---|---|
| D1 | D0 | *g* Range |
| 0 | 0 | ±2 *g* |
| 0 | 1 | ±4 *g* |
| 1 | 0 | ±8 *g* |
| 1 | 1 | ±16 *g* |

Below is a screenshot of the home page of the Processing application made for this project. This Processing application makes use of some new techniques learned for this project which help make the application look better. These include the implementation of custom fonts found online and also the ability to show PNG images designed in Adobe Illustrator. The rocket on the home page is a good example of a simple stylistic element designed in Illustrator, but the graphs seen on their respective pages are also produced in Illustrator and are much easier to create there and import to processing rather than attempting to draw the entire graphs with grids in processing (especially dashed lines).




One incredibly useful building-block learned during this project was the implementation of Java try and catch statements. Since there were many times when developing where there was no serial device plugged in it was necessary to include the statement below to stop the Processing sketch from crashing when opened.

```
try {
myPort = new Serial(this, "/dev/cu.usbserial-A50285BI", 9600);
//starts the serial communication
myPort.bufferUntil('\n');
} catch (RuntimeException e) {
  println("No Serial Connection! Live Data Tracking is Unavailable");
  serialConnection = false;
}
```

## Parts

The parts used for this project are shown to the right. Since a portion of the hardware must operate on the rocket while in flight, it requires a mobile power-source. Luckily, Nanos accept up to 12 V as an input on the Vin pin even though they operate themselves at only 5 V. This makes use of the onboard voltage regulator, which allows this project to use a simple 9 V battery to power the rocket-side.

| Parts Table | |
|---|---|
| Arduino Nano | 2 |
| ADXL345 Accelerometer | 1 |
| HC-12 RF Transceiver | 2 |
| 9 V Battery | 1 |
| Model Rocket | 1 |

## Code

The full code for this project can be found on the project GitHub:
https://github.com/Liam-McCartney/Hardware/tree/main/Rocket%20Telemetry

Transmitter Code:

```
#include <Wire.h>
#include <SoftwareSerial.h>
//SoftwareSerial is used to not impede on Serial Monitor

SoftwareSerial HC12(10,11);
//Create the HC12 Object

#define ADXL345 0x53
//Define Address

float X_out, Y_out, Z_out, rollF, pitchF;
//variables

void setup() {
  Serial.begin(9600);
  HC12.begin(9600);
  Wire.begin();

  //Set ADXL345 in measuring mode vvvv
  Wire.beginTransmission(ADXL345);
  Wire.write(0x2D);
  //Talk to power/control Register
  Wire.write(1 << 3);
  //Measure enable bit
  Wire.endTransmission();
  delay(10);

  //X-axis
  Wire.beginTransmission(ADXL345);
  Wire.write(0x1E);  //X-axis offset register
  Wire.write(-2);
  Wire.endTransmission();
  delay(10);
  //Y-axis
  Wire.beginTransmission(ADXL345);
  Wire.write(0x1F);  //Y-axis offset register
  Wire.write(-2);
  Wire.endTransmission();
  delay(10);

  //Z-axis
  Wire.beginTransmission(ADXL345);
  Wire.write(0x20);  //Z-axis offset register
  Wire.write(5);
```

```
  Wire.endTransmission();
  delay(10);

  Wire.beginTransmission(ADXL345);
  Wire.write(0x31);
  //Scale register
  Wire.endTransmission();

  Wire.beginTransmission(ADXL345);
  Wire.write(1 << 1);
  //Set the scale
  Wire.endTransmission();
}

void loop() {
  // === Read acceleromter data === //
  Wire.beginTransmission(ADXL345);
  Wire.write(0x32);
  //First data register
  Wire.endTransmission();
  Wire.requestFrom(ADXL345, 6);
  //6 total data registers
  X_out = (Wire.read() | Wire.read() << 8);  //X-axis value
  Y_out = (Wire.read() | Wire.read() << 8);  //Y-axis value
  Z_out = (Wire.read() | Wire.read() << 8);  //Z-axis value
  X_out = X_out / 64;
  Z_out = Z_out / 64;
  Y_out = Y_out / 64;
  //scale according to datasheet

  float roll = atan(Y_out / sqrt(pow(X_out, 2) + pow(Z_out, 2))) * 180 / PI;
  float pitch = atan(-1 * X_out / sqrt(pow(Y_out, 2) + pow(Z_out, 2))) * 180 / PI;
  //Roll and pitch formulas from online papers

  //Low-pass filter
  rollF = 0.94 * rollF + 0.06 * roll;
  pitchF = 0.94 * pitchF + 0.06 * pitch;
  //I though this was cool when I saw someone else doing it online so I did it here

  float acceleration = sqrt(pow(X_out, 2) + pow(Y_out, 2) + pow(Z_out, 2)) - 1;
  //Find physics-acceleration

  if (Z_out < 0) acceleration = -acceleration;
  //Make sure it has the correct sign

  HC12.print(rollF);
  HC12.print("/");
  HC12.print(pitchF);
  HC12.print("/");
  HC12.println(acceleration);
  //Send the data
}
```

## Media

Project Video: https://youtu.be/TYdosPPh7lc
Project GitHub: https://github.com/Liam-McCartney/Hardware/tree/main/Rocket%20Telemetry

Rocket With Electronics Visible

## Reflection

I am going to keep the reflection for this project pretty short: I think that while this project certainly took a turn toward software rather than hardware, I still accomplished the key functionality that I set out for. While I have not yet had the opportunity to launch the rocket, it definitely will happen at some point. I have already put in all the effort and I would like to see the project through. I will probably upload a simple video about it, and put the link on the GitHub to find later.

On to the bigger picture. This project is the last of them all. The final time writing in my DER. It is a key milestone for me. On one hand, I am done; I finished the ACES program and have many amazing projects and memories to look back on, but on the other hand I am sad that my time in Hardware and at high school in general is coming to an end. However, I know that everything I have learned over the past three years was 100% worth all the late nights and close deadlines, and I am excited to continue my learning next year and beyond. While it is near the time to move on to the future, I know that I will always remember the time that I spent in the DES and of course the person who made this all possible. Thank you Mr. D'Arcy.