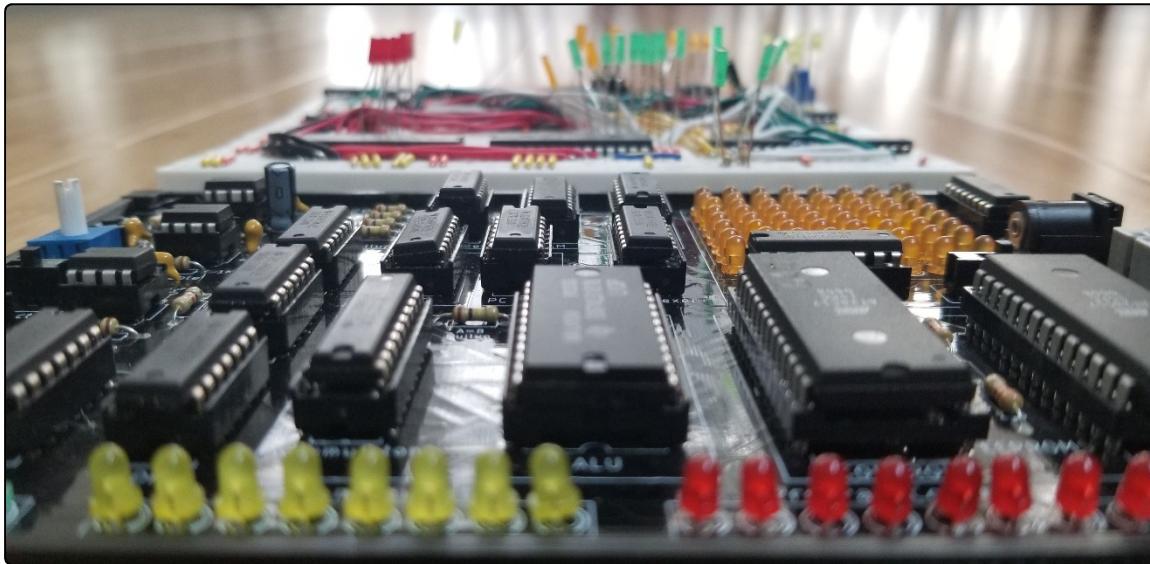


DESIGN ENGINEERING REPORT



Course: Computer Engineering Technology
Code: TEL3M, TEI3M, ICS4U
Instructor: C. D'Arcy
Author: Daniel Raymond
Date: Thursday, May 30, 2019

Table of Contents

<u>PROJECT 1. SOLDERING BASICS</u>	<u>1</u>
PURPOSE	1
REFERENCE.....	1
PROCEDURE.....	1
MEDIA.....	2
CONCLUSION.....	2
<u>PROJECT 2. AUTOMATIC NIGHT LIGHT.....</u>	<u>3</u>
PURPOSE	3
REFERENCE.....	3
THEORY	3
PROCEDURE	3
MEDIA.....	4
CONCLUSION.....	4
<u>PROJECT 3. DIGITAL (TRANSISTOR-BASED) LOGIC GATES</u>	<u>5</u>
PURPOSE	5
REFERENCE.....	5
<u>PROJECT 4. SHORT ISP: DIGITAL THERMOMETER</u>	<u>7</u>
PURPOSE	7
REFERENCE.....	7
THEORY	7
PROCEDURE	7
MEDIA.....	8
CONCLUSION.....	8
<u>TEI3M.....</u>	<u>9</u>
<u>PROJECT 5. TRAFFIC LIGHT ASSEMBLY AND TESTING</u>	<u>11</u>
PURPOSE	11
REFERENCE.....	11
PROCEDURE	11
CODE	11
MEDIA	12
CONCLUSION.....	12
<u>PROJECT 6. DIGITAL AND ANALOG: READING AND WRITING</u>	<u>13</u>
PURPOSE	13
REFERENCE.....	13

PROCEDURE	13
CODE	13
MEDIA	15
CONCLUSION.....	15

PROJECT 7. DESIGN CHALLENGE 1.....17

PURPOSE	17
REFERENCE.....	17
ACKNOWLEDGMENTS	17
PROCEDURE	17
MEDIA	17
CONCLUSION.....	17

PROJECT 8. POV: DUAL 7 SEGMENT DISPLAY19

PURPOSE	19
REFERENCE.....	19
PROCEDURE	19
CODE	20
MEDIA	21
CONCLUSION.....	21

PROJECT 9. DIY 8X8 LED MATRIX23

PURPOSE	23
REFERENCE.....	23
THEORY	23
PROCEDURE	23
CODE	24
MEDIA	26
CONCLUSION.....	26

PROJECT 10. FREQUENCY SPECTRUM ANALYZER.....27

PURPOSE	27
REFERENCE.....	27
PROCEDURE	27
CODE	28
MEDIA	29
CONCLUSION.....	29

PROJECT 11. SHORT ISP: LIGHT-BASED ALARM CLOCK.....31

PURPOSE	31
REFERENCE.....	31
THEORY	31
PROCEDURE	32

CODE	33
MEDIA	36
CONCLUSION.....	36

PROJECT 13. PRINTED CIRCUIT BOARD I: RTC CLOCK..... 37

PURPOSE	37
MEDIA	37

PROJECT 14. PRINTED CIRCUIT BOARD II: RTC CLOCK 39

PURPOSE	39
REFERENCE.....	39
THEORY	39
PROCEDURE	39
CODE	40
MEDIA	41
CONCLUSION.....	41

PROJECT 15. I2C VELOCITY DATA LOGGER 43

PURPOSE	43
REFERENCE.....	43
THEORY	43
PROCEDURE	43
CODE	44
MEDIA	46
CONCLUSION.....	46

PROJECT 16. MEDIUM ISP: BLUETOOTH PHOTO FRAME 47

PURPOSE	47
REFERENCE.....	47
THEORY	47
PROCEDURE	48
CODE	49
MEDIA	50
CONCLUSION.....	50

ICS4U: AVR OPTIMIZATION 51

PROJECT 17. CHUMP..... 53

PART A. THE CODE	53
PURPOSE	53
THEORY	53
CODE	53
REFLECTION.....	53

PART B. THE CLOCK	54
PURPOSE	54
REFERENCE.....	54
THEORY	54
PROCEDURE	54
MEDIA	55
REFLECTION.....	55
PART C. ALU: THE ARITHMETIC AND LOGIC UNIT.....	56
PURPOSE	56
REFERENCE.....	56
THEORY	56
PROCEDURE	57
MEDIA	58
REFLECTION.....	58
PART D. THE ROMs.....	59
PURPOSE	59
REFERENCE.....	59
THEORY	59
PROCEDURE	60
MEDIA	61
REFLECTION.....	61
PART E. THE COMPLETED PROCESSOR.....	62
PURPOSE	62
REFERENCE.....	62
THEORY	62
PROCEDURE	63
MEDIA	64
REFLECTION.....	64
<u>PROJECT 18. SHORT ISP: VIDEO TURNTABLE.....</u>	<u>65</u>
PURPOSE	65
REFERENCE.....	65
THEORY	65
PROCEDURE	66
CODE	67
MEDIA	69
REFLECTION.....	69
<u>PROJECT 19. KEYPAD STORAGE UNIT.....</u>	<u>71</u>
PURPOSE	71
REFERENCE.....	71
THEORY	71
PROCEDURE	72

CODE	72
MEDIA	74
REFLECTION.....	74

PROJECT 20. ROTARY BCD SWITCH..... 75

PURPOSE	75
REFERENCE.....	75
THEORY	75
PROCEDURE	75
MEDIA	76
REFLECTION.....	76
CODE	77

PROJECT 21. KNIGHT RIDER 79

PURPOSE	79
REFERENCE.....	79
THEORY	79
PROCEDURE	80
MEDIA	80
REFLECTION.....	80
CODE	81

PROJECT 22. THE 4-WIRE DC FAN..... 83

PART A. RESEARCH..... 83	
PURPOSE	83
REFERENCE.....	83
THEORY	83
MEDIA	84
REFLECTION.....	84
PART B. EXECUTION..... 85	
PURPOSE	85
REFERENCE.....	85
PROCEDURE	85
THEORY	86
CODE	89
MEDIA	93
REFLECTION.....	93

PROJECT 23. MEDIUM ISP: SOUND CONTROLLED POWER BAR..... 95

PURPOSE	95
REFERENCE.....	95
PROCEDURE	95
THEORY	96

REFLECTION.....	97
CODE	98
MEDIA	99

PROJECT 24. LONG ISP: CHUMP EXTENDED 101

PURPOSE	101
REFERENCE.....	101
THEORY	101
PROCEDURE	104
CODE	104
MEDIA	105
REFLECTION.....	105

PROJECT 25. FLEX: SNAKE VIDEO GAME 107

PURPOSE	107
REFERENCE.....	107
PROCEDURE	107
THEORY	108
REFLECTION.....	109
CODE	110
MEDIA	120

Project 1. Soldering Basics

Purpose

The project for this section of the engineering report is to learn the basics of soldering which will help in most future projects.

Reference

Soldering Guide: <https://www.youtube.com/watch?v=pfbs182mvIc>

Soldering Tools: <http://darcy.rsgc.on.ca/ACES/TEL3M/images/Soldering.png>

Procedure

During set up, the solder station is set to 300°. The wire is put in through the metal hole (terminal) attached to the potentiometer, then bent around it. One claw of the ‘helping hands’ is holding the wire and the other is hold the end of the potentiometer to secure it.

The soldering pen is touched to the terminal of the potentiometer and the wire for three to five seconds in order to heat them up. The solder touches the heated metals and melts, fusing the two together.

Heat shrink tubing slides onto the wire and covers the soldered joint. A heat gun is used to heat the tubing and solidify it tightly around the joint. This procedure repeats for all the components and terminals.



Wire bent through terminal

Parts	Quantity
Wire(s)	7
Heat Shrink Tubing	4
Push Button (PBNO)	1
Push Button Normally Closed (PBNC)	1
Rotary Potentiometer	1

Media

Personal Soldering Video:

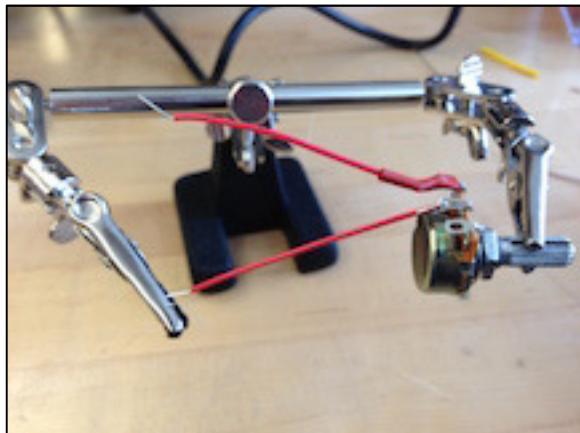
<https://www.youtube.com/watch?v=KrxHTNetOPc&feature=youtu.be>



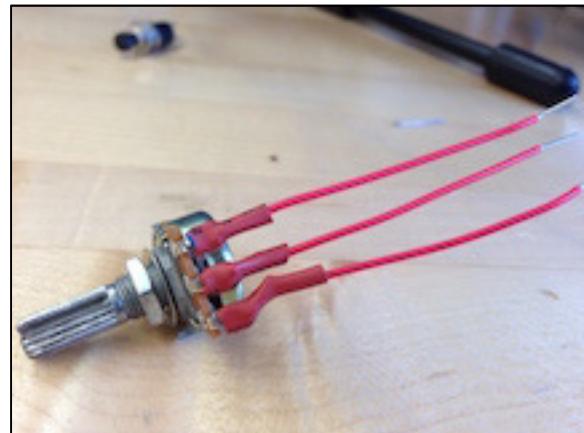
Push Button Normally Open (PBNO) 1



Variable Resistor (Rotary Potentiometer) 1



Potentiometer in the helping hands 1



Final product of the potentiometer 1

Conclusion

This project teaches the fundamentals of soldering wires to different components including potentiometers, PBNO's, and PBNC's.

Project 2. Automatic Night Light

Purpose

The goal of this project is to create a circuit that will automatically turn on an LED when there is very little light. This uses a custom PCB supplied by the DES.

Reference

Automatic Night Light Creation Video: <https://youtu.be/ROdQeRHcWwM>

Voltage Divider Analogy:

https://chem.libretexts.org/Core/Analytical_Chemistry/Electrochemistry/Voltage,_Amperage_and_Resistance_Basics

Theory

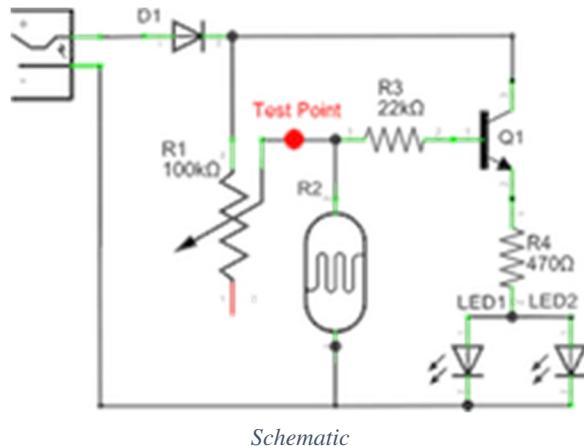
The variable voltage divider is well explained in the waterfall analogy (see “Voltage Divider Analogy” in the reference section). When the resistance of the photoresistor increases so that it is more than the $22\text{K}\Omega$ resistor, the electrons decide to go through the fixed resistor instead. This closes the transistor, allowing electrons to flow through the LEDs.

Procedure

The parts are assembled and arranged in the pattern of the schematic or fritzing diagram shown. This includes the soldered variable resistor from the first project.

The assorted parts are inserted into the circuit board. A switch is added to the circuit to reduce power consumption while inactive. The polarized components are put in the correct direction, and all the legs are bent so that each part is secured to the board. Each wire is soldered to the board.

Once this is done, the ends of the legs are clipped to ensure that there is no contact between any legs.



Schematic

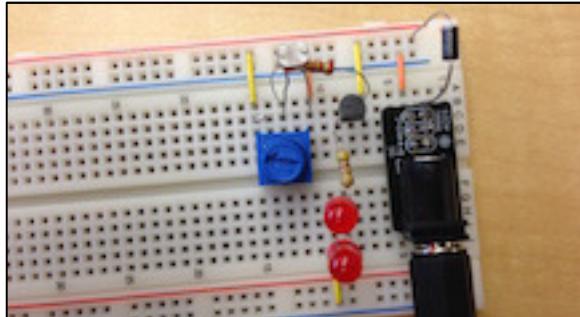
The inserts in the acrylic stand are popped out. The nut and washer are removed from the potentiometer as it is pushed through the center hole. The washer and nut are put on the potentiometer on the opposite side of the acrylic stand. The four smaller holes are aligned with the holes in the circuit board. The spacers are positioned in between the stand and the board. The screws secure the spacers, holding the circuit to the acrylic stand. A specially made knob slides on the potentiometer and any necessary final touches are made.

Data Gathering

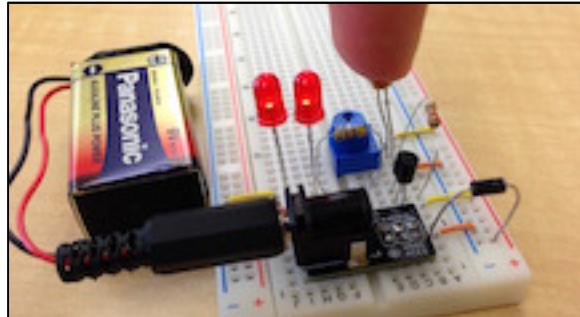
When the potentiometer (pot) has no resistance and there is plenty of light, the voltage at the ‘test point’ (see figure 1) is 8.14 V. When the light is reduced, the voltage increases to 8.23 V. When the pot is set to 50 K Ω with lots of light, the resulting voltage is 1.52 V. With very little light, there is 4.02V. When the pot is set to 100 K Ω with the same setup the results are 0.95 V and 2.92 V. This is because of there is a variable voltage divider. If the two resistors were equal, then the voltage at the test point would cut in half, as the light decreases, the second resistance increases less of the voltage is used in the first resistor, increasing the voltage at the test point.

Parts List	Quantity
NPN 3904 Transistor	1
470 Ω Resistor	1
22K/10K Ω Resistor	1
Variable Resistor	1
Photoresistor	1
Diode	1
LED	2
9V Battery	1

Media



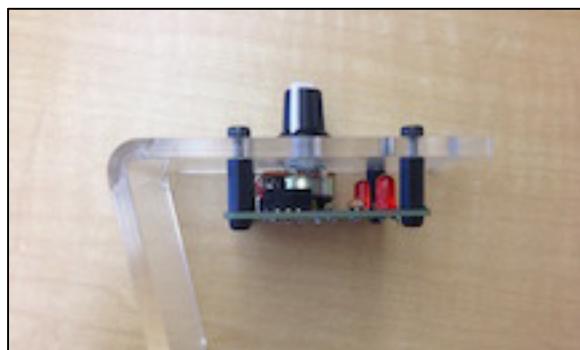
Prototype



Prototype – Lit Up



Soldering



Finished Product

Conclusion

The creation of the Automatic Night Light reinforced my knowledge of soldering from the last project. It also taught me how to read a schematic with confidence. The project was very enjoyable, and I wouldn't change anything about it.

Project 3. Digital (Transistor-Based) Logic Gates

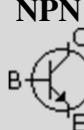
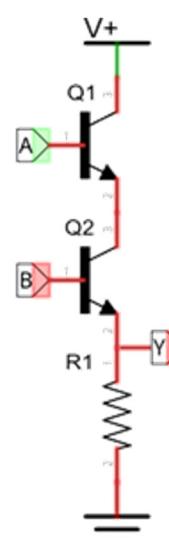
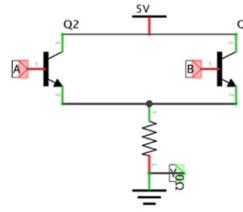
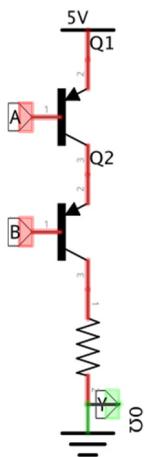
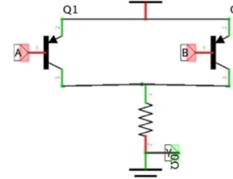
Purpose

The purpose of this project is to develop an understanding of logic using mechanical switches, transistors, and integrated circuits (ICs).

Reference

Description: <http://darcy.rsgc.on.ca/ACES/TEL3M/1617/TasksFall.html#Gates>

Fritzing: <http://fritzing.org/home/>

TRANSISTOR	SERIES	PARALLEL																														
NPN 	<p>Name: AND Symbol: </p> <table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Y</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table> 	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	<p>Name: OR Symbol: </p> <table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Y</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>1</td></tr> </tbody> </table> 	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Y																														
0	0	0																														
0	1	0																														
1	0	0																														
1	1	1																														
A	B	Y																														
0	0	0																														
0	1	1																														
1	0	1																														
1	1	1																														
PNP 	<p>Name: NOR Symbol: </p> <table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Y</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>0</td></tr> <tr> <td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table> 	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	<p>Name: NAND Symbol: </p> <table border="1"> <thead> <tr> <th>A</th><th>B</th><th>Y</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>1</td></tr> <tr> <td>0</td><td>1</td><td>1</td></tr> <tr> <td>1</td><td>0</td><td>1</td></tr> <tr> <td>1</td><td>1</td><td>0</td></tr> </tbody> </table> 	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Y																														
0	0	1																														
0	1	0																														
1	0	0																														
1	1	0																														
A	B	Y																														
0	0	1																														
0	1	1																														
1	0	1																														
1	1	0																														

Project 4. Short ISP: Digital Thermometer

Purpose

The project was designed turn a bicolour LED red or green, depending on if the ambient temperature was warm enough to wear a jacket. If the LED is red than a jacket should not be worn. If the LED is green, then the opposite is true.

Reference

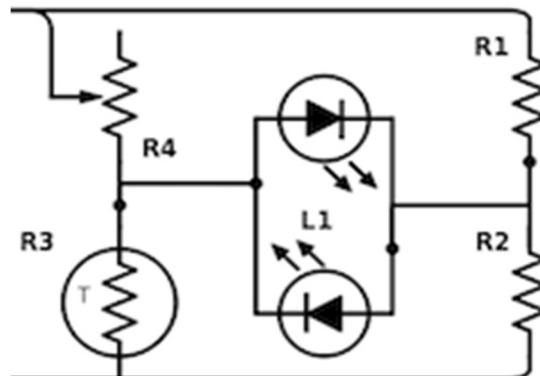
Digital Thermometer Creation Video: <https://youtu.be/vCjDfhSZqLA>

Voltage Divider Analogy:

https://chem.libretexts.org/Core/Analytical_Chemistry/Electrochemistry/Voltage,_Amperage_and_Resistance_Basics

Theory

Two voltage dividers are on either side of a bicolour LED. One side has fixed resistors while the other has a potentiometer and a thermistor. As the temperature increases, the resistance decreases and thus the voltage in the middle of the two resistors decreases. The voltage flows through the LED from the fixed resistor voltage divider turning the LED red. The opposite is true when the temperature decreases.



Procedure

There were multiple prototypes used in creating this project. The first idea was very similar to the Automatic Night Light, but an NTC thermistor replaced the LDR. The problem with this design was that the transistor is an analog component and would leak voltage through. As a result of this, the LED would not be on sometimes.

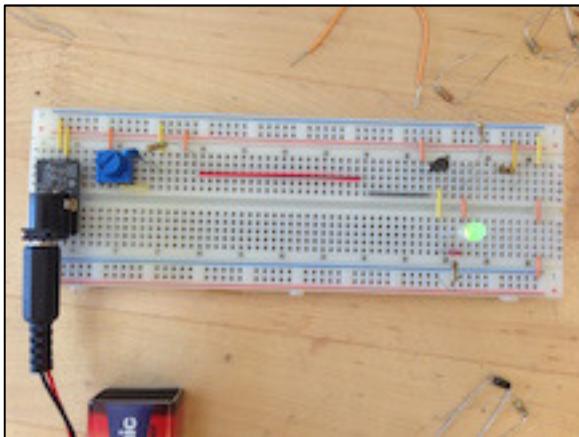
After a long time fiddling around with prototypes an op amp comparator circuit was decided on. It works by comparing the voltage given from a thermistor coming straight from

V^+ and the voltage from a voltage divider giving 6V. If the thermistor gives more than 6V, then the 741 IC outputs almost all 9V. If it doesn't then the IC outputs almost 0V. This goes to a resisted bicolour LED that has another voltage divider on the other leg. This makes it always the LED either red or green, but not off.

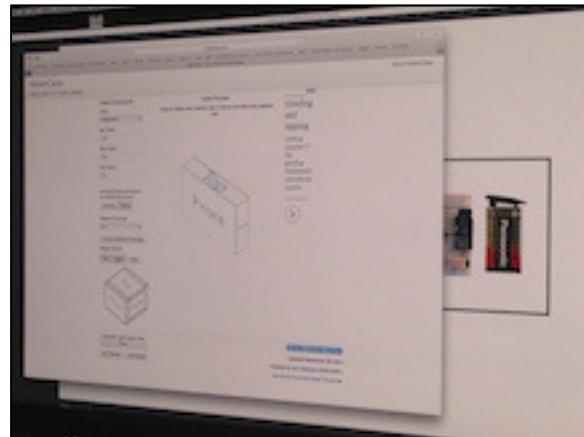
The next step was to solder the final working prototype to a solder-able breadboard. After this a casing was made. A piece of acrylic was laser cut into multiple pieces that fit together to form a box. The soldered circuit was then attached to the casing.

Parts	Quantity
NTC 10K Thermistor	1
Variable Resistor	1
741 IC	1
Wires	11
1KΩ Resistor	1
680Ω Resistor	1
470Ω Resistor	1
10KΩ Resistor	3
Bicolour LED	1
9V Battery	1

Media



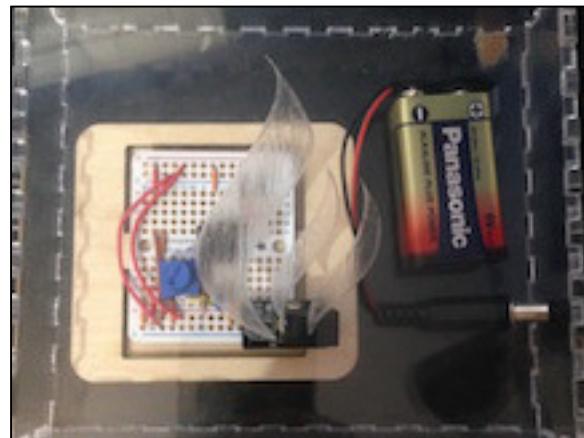
Original prototype



Designing the case



Constructing the case



Finished product

Conclusion

This project helped develop an understanding for voltage dividers and the 741 IC. It has a useful application and has put an emphasis on designing a finished product. Overall, this project was very enjoyable

TEI3M

Project 5. Traffic Light Assembly and Testing

Purpose

This goal of this project is to introduce Arduino coding, get familiar with appliances, and create a working prototype of a traffic light.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html#TrafficLightAssembly>

Arduino Language Basics: <https://www.arduino.cc/en/Reference/HomePage>

Procedure

The parts are assembled and inserted into the PCB board given. The LED's and the 470Ω resistor are soldered to the board and tested to ensure that they all are still working. The Traffic Light PCB is inserted into the Arduino board using the four-pin right-angle male header with the ground pin in the one marked 'ground'. The Arduino code is configured to correspond correctly with each LED in the setup function. Code is written to turn on each LED one after another.

Parts	Quantity
Traffic Light PCB	1
10mm LED (Red, Yellow, Green)	3
470Ω Resistor	1
Arduino UNO (ATMega328p)	1
USB 2.0 A to B cable	1

Code

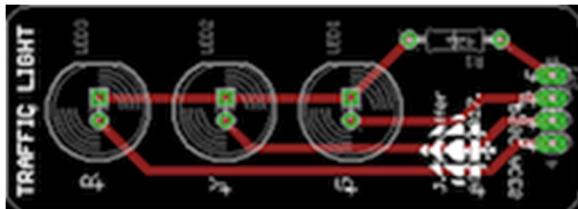
```
// Name    : Traffic Light
// Author  : Daniel Raymond
// Status   : Functional
byte red = 11;
byte yellow = 12;
byte green = 13;
int timer = 1000;
void setup() { // Tells the board to output at certain pins
    pinMode(red, OUTPUT);
    pinMode(yellow, OUTPUT);
    pinMode(green, OUTPUT);
}
void loop() { // Loop that makes the cycle continue forever
    digitalWrite(green,HIGH); // Turns the green LED on (High voltage)
    delay(timer); // Waits for an amount of time set by the 'timer'
    digitalWrite(green,LOW); // Turns the green LED off (Low voltage)

    digitalWrite(yellow,HIGH); // Turns the yellow LED on
    delay(timer/2); // Waits for half of the timer's set time
    digitalWrite(yellow,LOW); // Turns the yellow LED off

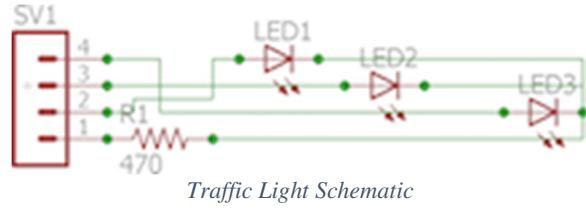
    digitalWrite(red,HIGH); // Turns the red LED on
    delay(timer); // Waits for the timer's set amount
    digitalWrite(red,LOW); // Turns the red LED off
}
```

Media

Personal Video: <https://youtu.be/7MSwACfHqUY>



Jasper's PCB Appliance



Traffic Light Schematic



Traffic Light with Green on High



Close-up of Pin Configuration

Conclusion

This project taught simple lines of code that are very applicable and fundamental to the use of Arduino. There is also an actual working prototype of a traffic light to show for it.

Project 6. Digital and Analog: Reading and Writing

Purpose

The purpose of this project is to have analog and digital reading and writing implemented to make a more realistic traffic light.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html#TrafficLightAssembly>

Arduino attachInterrupt Page: <https://www.arduino.cc/en/Reference/AttachInterrupt>

Procedure

The traffic is used alongside a bicolour LED that acts as a pedestrian crossing light. A button and an IR sensor have input pins connected to them. The code turns on the LED's for certain intervals of time depending on the inputs. For the PBNO, an interrupt command is used to ensure that the button press is measured. For the distance sensor, an equation converts the sensor's measurement into a readable distance. The equation was calculated from a voltage distance graph found in the [datasheet](#).

Parts	Quantity
Traffic Light	1
Arduino board (ATMega328p)	1
USB 2.0 a to b cable	1
PBNO	1
IR Distance Sensor (GP2D120x)	1
Wires	3
220Ω Resistor	1
Bicolour LED	1

`DigitalWrite()` is used to turn on all the LED's except for the red traffic LED.

`AnalogWrite()` is used to turn on the red traffic LED at a dimmed brightness.

`DigitalRead()` is used to measure whenever the PBNO is pressed.

`AnalogRead()` is used to measure the IR voltage, and then converted into a usable distance.

Code

```
// Name      : Advanced Traffic Light
// Author    : Daniel Raymond
// Status    : Functional

// Global variable setup here
uint8_t red = 11; // Output pin assignments
uint8_t yellow = 12;
uint8_t green = 13;
uint8_t pedRedPin = 5; // Pedestrian lights
uint8_t pedGreenPin = 6;

uint8_t sensorIRPin = A1; // Input pin assignments
uint8_t buttonPin = 2;

uint16_t car = 0; // converted value of the IR Sensor to cm
boolean pedestrian = LOW; // value to remember if the button has been pressed

uint16_t timer = 3000; // Amount of time a light will stay on
```

```
float gChange = 1; // factors for how long a light will stay on
float rChange = 1;

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

    pinMode(red, OUTPUT); // Assigns traffic light pins
    pinMode(yellow, OUTPUT);
    pinMode(green, OUTPUT);

    pinMode(pedRedPin, OUTPUT); // Assigns pedestrian light pins
    pinMode(pedGreenPin, OUTPUT);

    pinMode(sensorIRPin, INPUT); // Assigns sensor pins here
    pinMode(buttonPin, INPUT);
    attachInterrupt(digitalPinToInterruption(buttonPin), pressing, RISING);
    // Sets up button to always be read; to "interrupt" the loop
}

void loop() {
    car = 2914 / (analogRead(sensorIRPin) + 5) - 1;
    // Converts reading to an approximate distance in cm
    delayMicroseconds(1);

    if (car < 10)
        // If car is close to the intersection: make the green last longer
        gChange = 2;
    else if (pedestrian == HIGH)// Determines if pedestrian is at the crosswalk
        gChange = 0.5;
    else
        gChange = 1; // If neither, the light will stay on the normal amount

    lightOn(green, timer * gChange); // Turns the green led on, waits, then off
    lightOn(yellow, timer >> 1); // Turns the yellow led on, waits, then off

    if (pedestrian == HIGH)
        // Determines if a pedestrian is at the crosswalk: makes the red last longer
        rChange = 2;
    else if (car < 10)
        // Determines if car is close to the intersection: makes red shorter
        rChange = 0.5;
    else
        rChange = 1;
    // If neither, the light will stay on the normal amount of time

    pedestrian = LOW; // Resets the button

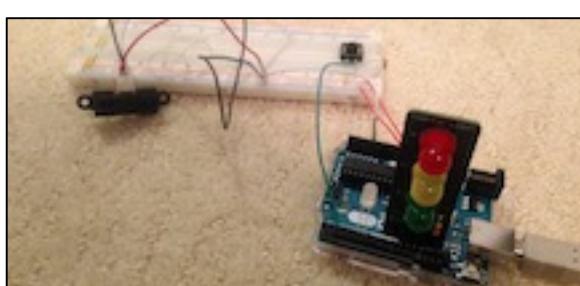
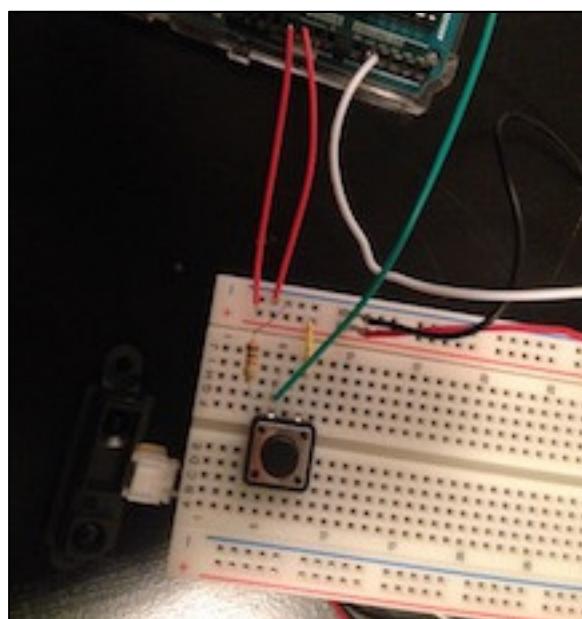
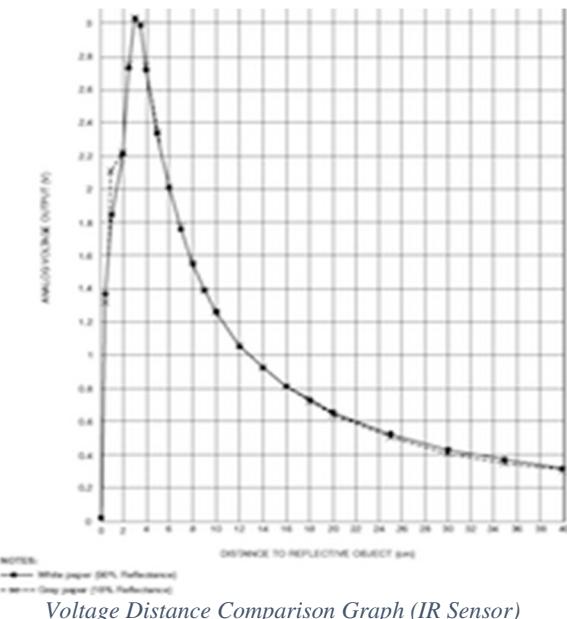
    analogWrite(red, 8); // Turns on a dulled red led
    digitalWrite(pedRedPin, LOW);
    digitalWrite(pedGreenPin, HIGH); // Turns pedestrian light green
    delay(timer * rChange);
    digitalWrite(red, LOW); // Turns off the red led
    digitalWrite(pedGreenPin, LOW); // Turns pedestrian light red
    digitalWrite(pedRedPin, HIGH);
} // End Loop
```

```
// Function that turns on the main traffic lights
void lightOn(uint8_t pin, uint16_t duration) {
    digitalWrite(pin, HIGH);
    delay(timer);
    digitalWrite(pin, LOW);
} // End LightOn

// function that is called whenever the button is pressed
void pressing() {
    pedestrian = HIGH;
} // End pressing
```

Media

Personal Video: https://www.youtube.com/watch?v=brW-E_132v4&feature=youtu.be



Conclusion

This project strengthened a foundation in using reading and writing statements. The addition of external components on a bread board was also a new concept that was learned.

Project 7. Design Challenge 1

Purpose

The purpose of this project was to ensure a solid grasp of the ViaCAD program and develop a prototype case for future ACES to use.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html#DesignChallenge1ER>

Acknowledgments

A huge thanks to Mr. Paul Elia, who was a major mentor for the students who undertook this project. He taught students how to work the ViaCAD program and answered a flourish of questions. Kreher Fiset-Algarvio and Darius Dadyburjor were also a major help with this project, who stayed in the design studio for hours printing the cases. Finally, thanks to *Sawdust & Noise* for precisely laser cutting the acrylic tops for the cases.

Procedure

A perma-proto full-size breadboard case was created to allow more space for people to create their projects. Stacking half-size boards was an option that was avoided because it needlessly complicates things. The lid was originally designed to have a slit in which it slides into, but it stays in place without it. The dimensions were made such that, when the lid is inserted, it remains inserted with no slit required. The force of the case squeezing the lid ever so slightly keeps it in place. This, by itself, makes the lid difficult to remove, so a thumb hole was added to help with this problem. There is also a small opening in the side of the case to allow a power source to be inserted without much trouble.

Media



"RSGC ACES" engraved onto the front



Final Prototype

Conclusion

This project finalized an understanding for the ViaCAD program. A finished product was created, which could be added to the school store for future students to use.

Project 8. POV: Dual 7 Segment Display

Purpose

The purpose of this project was to maximize the hardware capabilities by alternating signals on the same wire. This took the form of lighting up two seven-segment displays with only eight Arduino pins and an input for it to display.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html#POV1>

IR Sensor Datasheet: https://www.sparkfun.com/datasheets/Sensors/Infrared/gp2y0a02yk_e.pdf

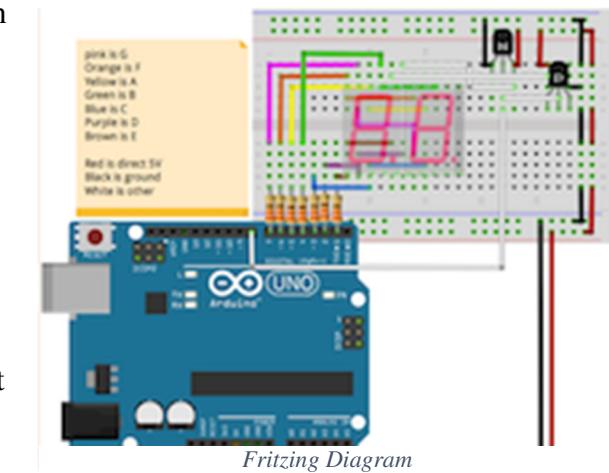
Procedure

The pins of the dual seven-segment display are connected so that 'a1' is connected to 'a2', 'b1' is connected to 'b2', and so on. The pins are then resisted and attached to Arduino pins one to seven so that they correspond to segments 'g', 'f', 'a', 'b', 'c', 'd', and 'e' respectively. The ground of one of the seven-segment displays is connected to the collector pin of an NPN transistor, and the other ground is connected to the emitter pin of a PNP. Both base pins of each of the transistors are connected to a pin that emits a square wave so the two oscillate with alternate signals. This turns each seven-segment display on and off in an alternating pattern so that, when one is on the other is off.

An infrared distance sensor is wired to ground, voltage, and an analog input pin to provide the data that is displayed on the display. The specific IR sensor that was selected measures from 20cm to 150cm.

An array of bytes is configured in such a way that each bit will light up a certain LED in the seven-segment display, and each byte shows a digit. A value is read in by the IR sensor and is converted to centimetres, then restricted to a two-digit maximum. When the information is sent to the displays, the code turns one display off so that only one of them shows the resulting number. Different information is sent, and immediately the other display is turned on to show that information. This oscillation occurs so rapidly that there is a "persistence of vision" effect where it appears both displays are on at the same time but displaying different digits.

Parts	Quantity
602MK2 Dual seven-segment display	1
Arduino board (ATMega328p)	1
USB 2.0 a to b cable	1
2Y0A02 Distance sensor	1
NPN 3904 Transistor	1
PNP 3906 Transistor	1
220Ω Resistor network	2
10KΩ resistor	3



Code

```

// Purpose: To display the distance coming from a IR sensor in centimeters on
// two seven-segment displays.
// Name    : Distance Display
// Author  : Daniel Raymond
// Status   : Functional

// Global variable setup here
uint8_t dist;           // Converted analog input
uint8_t cnst;           // Constrained dist
uint8_t sensePin = A3; // Sensor pin
uint8_t tens;            // Left digit displayed
uint8_t ones;            // Right digit displayed
uint8_t base = 10; // Current base (in base 10)
uint8_t totP = 50; // Total Period: in milliseconds
uint8_t osiP = 5; // Oscillation Period: must be lower than totP

// gfabcde.
uint8_t display[10] = {
  0b01111110, // zero
  0b00011000, // one
  0b10110110, // two
  0b10111100, // three
  0b11011000, // four
  0b11101100, // five
  0b11101110, // six
  0b00111000, // seven
  0b11111110, // eight
  0b11111000, // nine
};

void setup() { // Runs once at the beginning
  DDRD = 0xFE; // pinMode for the 7 segment pins
  DDRB = 0x21; // pinMode for the oscillating pin and the power pin
}

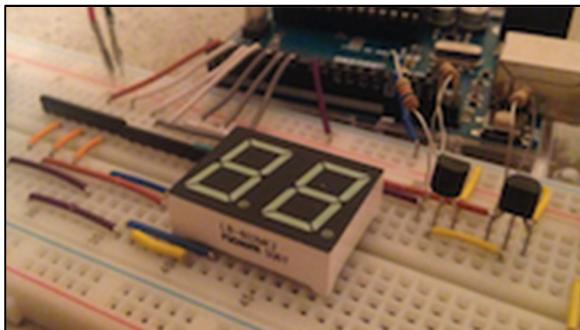
void loop() { // Loop that makes the cycle continue forever
  dist = 9462 / (analogRead(sensePin) - 18); // Converts analogRead to cm
  delayMicroseconds(1);
  cnst = constrain(dist, 20, 99); // Constrains dist

  tens = cnst / 10; // Isolates the tens digit
  ones = cnst % 10; // Isolates the ones digit
  // Loops oscillating #'s for
  // totP regardless of osiP
  for (int x = 0; x < totP*2/osiP; x++) {
    PORTB = 0x00; // Turn off left 7-segment
    PORTD = display[ones]; // Sends information for right 7-segment
    delay(osiP);
    PORTB = 0x20; // Turns off right 7-segment
    PORTD = display[tens]; // Sends information for left 7-segment
    delay(osiP);
  }
}

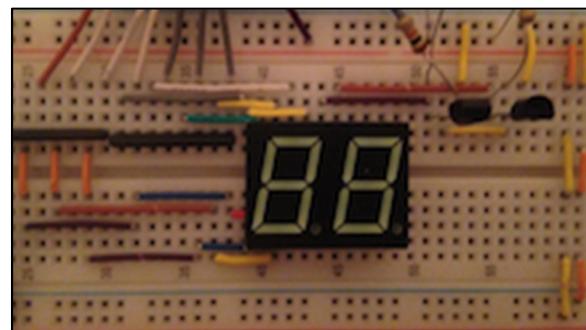
```

Media

Personal Video: <https://www.youtube.com/watch?v=o28yVVWjSD0&feature=youtu.be>



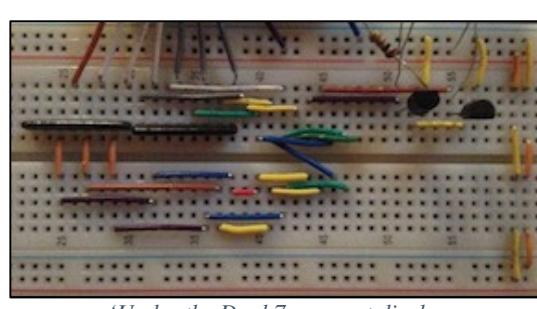
'Trimetric Display View'



'Top Wiring View'



'Circuit in Action'



'Under the Dual 7-segment display'

Conclusion

Overall, this project was a huge success. The oscillation of the ground pins on the dual seven-segment display was a very clever and unique way to reduce pin usage. It has taught the class to be mindful of how their Arduino is wired and to help raise the potential limit of what can be achieved. Mid-level programming was also introduced in this project and is another great advantage for coders to understand as it allows direct access to the data that is stored.

Project 9. DIY 8x8 LED Matrix

Purpose

The purpose of this project is to create an LED matrix with anything on it. This is very open ended, so in this design the matrix acts as a timer in which the user can enter a time in seconds, and the LED matrix will begin to light up row by row. When it is completely lit up, the time is finished, and it will start blinking until you turn it off via a button.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html#DIY>

Timer Interrupt Library: <http://playground.arduino.cc/Code/Timer1>

Theory

Since LED matrices have all the anode legs of each row connected, and all the cathode legs of each column connected, creating diagonal lines are impossible to do at the same time. To achieve this however, rows must be alternated incredibly fast using a persistence of vision trick so that the illusion of a diagonal line is created. This trick can be used to show the individually incrementing line, and then the previously filled rows which can be all displayed at the same time. Using this technique, the desired visual output can be achieved. To create the correct intervals of time, timer interrupt statements are used to determine how long each LED should be on for and when to execute the next command.

Parts	Quantity
DIY 8x8 LED Matrix (3mm)	1
Arduino board (ATmega328p)	1
USB 2.0 a to b cable	1
PBNO	1
Shift Register (SN74HC595)	2

Procedure

To begin this arduous task, 64 three-millimetre LEDs are bent so that their anode and cathode legs make a right angle perpendicular to the LED head. After doing this for all the LEDs, rows of eight LEDs are soldered together using a previously 3D printed rig to ensure the correct spacing. Once eight rows are created, the remaining cathode legs are soldered from each of the rows to form the eight by eight matrix (See figure 1). Heat shrink is used on each connection to ensure that none of the anode wires touch any of the cathode wires. After this, a cover is 3D printed to keep the correct spacing of the LEDs during use and to improve the aesthetic of the design.

To wire up the matrix, two shift registers are used; one for the cathode legs and one for the anode legs. The data, clock, and latch pins are separately wired to the Arduino board. Since the pins of the board are internally resisted, the outputs of the shift registers do not need to be tied to resistors. As per usual, the grounds are connected to the Arduino pin and power is applied.

Code

```
// Name : Custom Matrix Timer
// Author : Daniel Raymond
// Status : Functional

// Pins
uint8_t rowLatch = 2;
uint8_t rowData = 3;
uint8_t rowClk = 4;
uint8_t colLatch = 5;
uint8_t colData = 6;
uint8_t colClk = 7;

uint8_t btn = 13;

// Global variable setup
uint8_t cRow = 0; // Current changing row
uint8_t bRows = 0; // How many rows below should be lit up

uint16_t timer = 0;
double input;

#include <TimerOne.h>

void setup() {
    Serial.begin(9600);
    DDRD = 0b11111100; // Sets up the shift register pins

    // Timer that interrupts the code 1/100th of a second to add one to timer
    Timer1.initialize(10000);
    Timer1.attachInterrupt(second);
}
```

```
void loop() {
    if (Serial.available() > 0) {
        input = Serial.parseInt(); // Reads in time in seconds

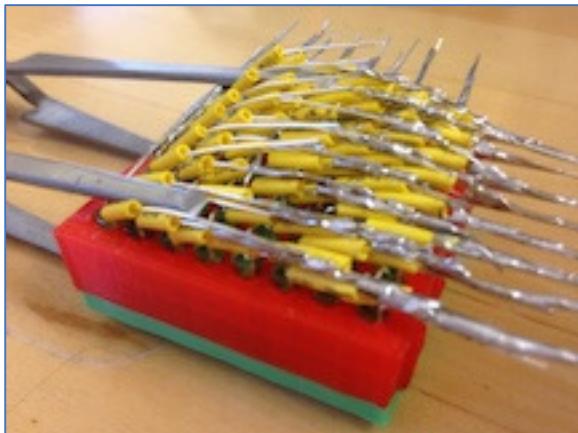
        for (uint16_t row = 0; row < 7; row++) {
            for (uint16_t col = 0; col < 7; col++) {
                // Changes led only when time is 1/49th of the way done
                while (timer < ((input / 49) * 100)) {
                    for (uint8_t cycle = 0; cycle < 7; cycle++) {
                        display(cRow, ~(1 << row)); // Displays the running row
                        display(255, ~(bRows)); // Displays rows below the running row
                    }
                }
                timer = 0; // Resets timer
                cRow <= 1;
                cRow++; // Adds one to the current row
            }
            bRows += 1 << row; // Creates one more full row below the current
            cRow = 0; // Resets the current row
        }
        while (digitalRead(btn) == 0) { // Flashes when timer is done
            display(255, 0);
            delay(250);
            display(0, 0);
            delay(250);
        }
        bRows = 0; // Resets full row
    }
    display(0, 0); // Wipes screen
} // End Loop

void display(uint8_t rowInfo, uint8_t colInfo) {
    PORTD = 0; // Closes the latch
    shiftOut(rowData, rowClk, LSBFIRST, rowInfo);
    shiftOut(colData, colClk, LSBFIRST, colInfo);
    PORTD = 0xFF; // Opens the latch
}

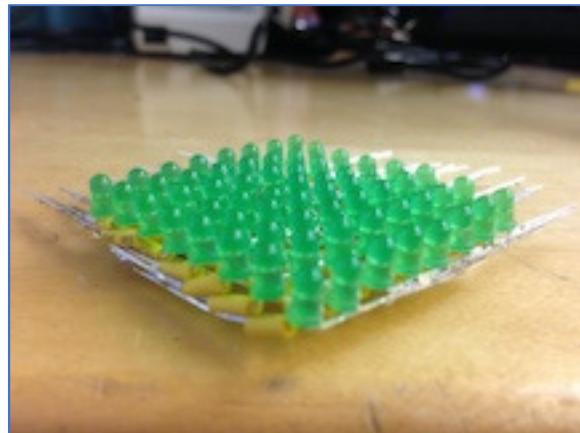
void second() {
    timer++; // Timer increments
}
```

Media

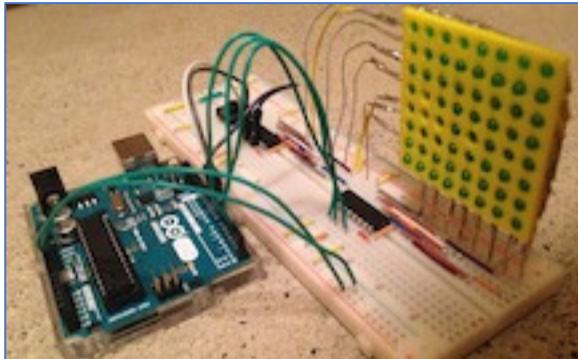
Personal Video: <https://youtu.be/gGvHRAdu4Y8>



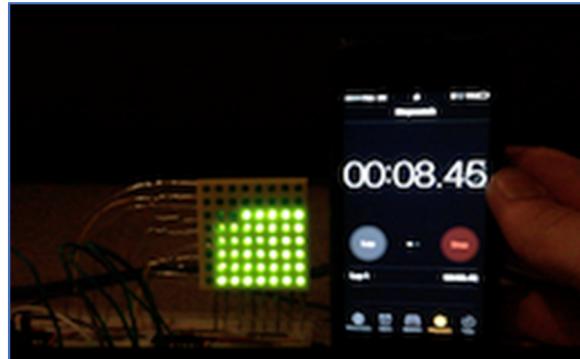
LED Finished Soldering



Trimetric LED View



Completed Circuit



Circuit in Action

Conclusion

This project took up a lot of time. The ambitious idea to solder the 8x8 instead of the 4x4 was, while enjoyable, very intensive. I was in the Design Engineering Studio after school four of the five days outside of class adding up to a total of eight hours, not including work at home. While I enjoyed the project, most of the time was spent soldering. After all the time spent on the project, it is disappointing that one row and one column didn't work even though none of the LEDs were burnt out and all of them were facing the correct way. It remains a mystery. Overall, it was a fun project that showed the complexity of even the simplest parts we've used. It makes me appreciate the parts and the fact that we don't have to make them ourselves.

Project 10. Frequency Spectrum Analyzer

Purpose

The goal of this project is to wire an 8x8 LED matrix to display the decibel level of seven unique frequency spectrums using the ATtiny84 microcontroller off board and the MSGEQ7 frequency analyzing chip.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html#FSA>

Equalizer Datasheet: <https://www.sparkfun.com/datasheets/Components/General/MSGEQ7.pdf>

Procedure

To begin this project, the ATtiny84 is wired to two shift registers using three output pins. The clocks of the shift registers are wired together as well as the two latches. For the data pins, the overflow pin of one of the shift registers goes into the data of the other. In this way, we can use two functioning shift registers while only using three pins of the microcontroller. One of the shift registers' output pins are wired to each of the cathode column pins and the other is connected to the anode row pins. To test that this setup works properly, data is hardcoded into an array which is then shifted out using the persistence of vision method (POV).

Parts	Quantity
8x8 LED Matrix (3mm)	1
ATtiny84	1
Pocket AVR Programmer	1
Shift Register (SN74HC595)	2
7 Band Graphic Equalizer (MSGEQ7)	1
Electret Microphone (MAX9814)	1
TRS audio jack (3.5mm)	1
Capacitor	4
Resistor	1

The next iteration of this project is the addition of the dual inputs. For this, a microphone and a TRS audio jack are used with a switch to determine which of the inputs is to be in use. Since the audio jack has left audio (tip) and right audio (ring) but only one input is needed, the two can be wired together and sent to the switch. To interpret the sine wave of voltage that either of the two inputs are sending, an equalizer chip is used. This chip converts its input into seven discrete 'bands' of data that correspond to seven different frequencies (See *MSGEQ7 Frequency Response*). To send this serial data, a strobe, reset and output pin are used. The data pin of the equalizer chip must be sent to an analog-input pin of the microcontroller.

Code is written to read in the data sent from the equalizer chip by manipulating the reset and strobe pins. Since the data that is read is either from 0 to 1023 due to the 10 bit analog to digital converter within the microcontroller, and a total of eight possible decibel levels for the LEDs is required, the raw data can be bit shifted right seven times to give eight possible outcomes. This means that any other bits other than the top three most significant ones are useless. This value from one to eight is then used as the index for another hardcoded array which has eight incrementing values that are all one less than a power of two. This then is shifted onto the column pins of one of the shift registers using POV. This displays our required seven bands.

Code

```
// Name      : Frequency Spectrum Analyzer
// Author    : Daniel Raymond
// Status    : Functional

// Global variable setup here
const uint8_t data = 0;
const uint8_t ltch = 1;
const uint8_t clk = 2;

const uint8_t strb = 8;
const uint8_t reset = 9;
const uint8_t inPin = A7;..

uint8_t spectrum [8];
const uint8_t values [8] = {0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF};

void setup() { // Runs once at the beginning
  DDRA = 0x07;
  DDRB = 0x06;
}

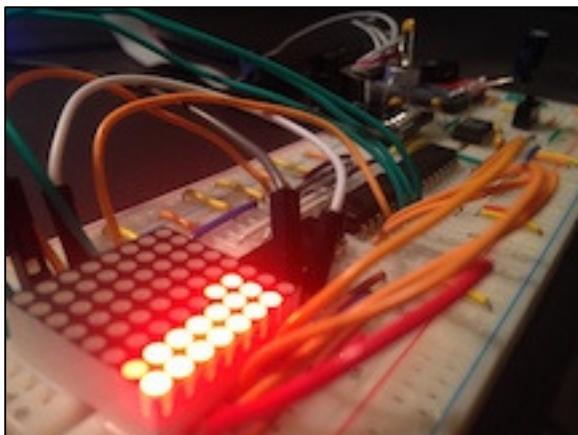
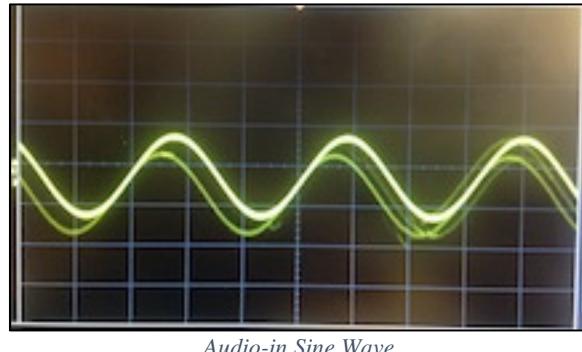
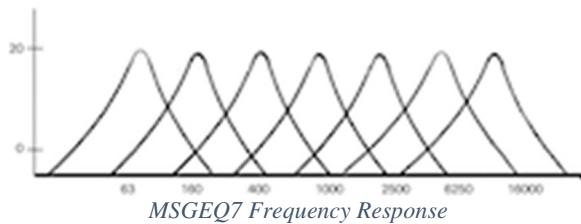
void loop() { // Loop that makes the cycle continue forever
  // Reset input for the MSG
  digitalWrite(reset, HIGH);
  digitalWrite(reset, LOW);

  for (uint8_t i = 0; i < 7; i++) {
    // Sets MSG latch to be able to read in
    digitalWrite(strb, LOW);
    // Uses 3 most significant bits to tell how many LEDs to turn on
    spectrum[i] = values[analogRead(inPin) >> 7];
    digitalWrite(strb, HIGH); // Sends info

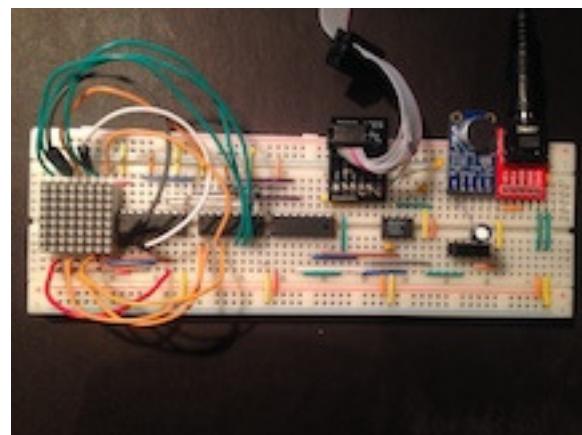
    digitalWrite(ltch, LOW);
    shiftOut(data, clk, LSBFIRST, spectrum[i]); // V+ data (col)
    shiftOut(data, clk, MSBFIRST, ~(1 << i)); // GND data (row)
    digitalWrite(ltch, HIGH);
  }
}
```

Media

Personal Video: <https://youtu.be/lyGmcqZ0vVI>



Trimetric LED View



Complete Circuit

Conclusion

I have learned a lot about audio to voltage conversion and its simplicity. Taking the sine wave of the physical audio vibrations and converting it to a sine wave of voltage is deceptively simple. This project was a supposed to be a simple concept for learning about audio devices and letting us work on our ISP, but this was not the case. The class spent quite a long time trying to create this device. Although I feel as though I learned a lot from this project, it was also quite frustrating to wire it.

Project 11. Short ISP: Light-Based Alarm Clock

Purpose

This independent study project (ISP) is designed to wake the user up using light with a settable alarm, all while showing the time. The clock must have a 3D printed case, ensuring the final design has the aesthetic of an actual product.

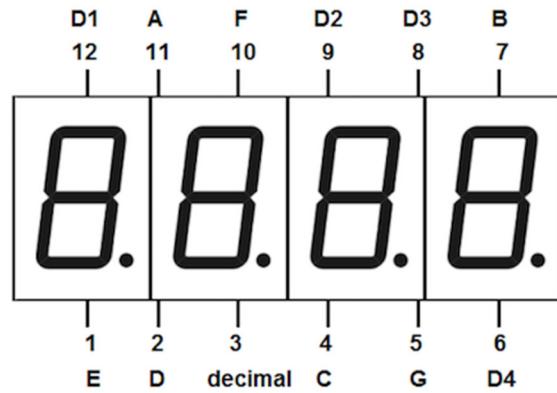
Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/ISPs.html>

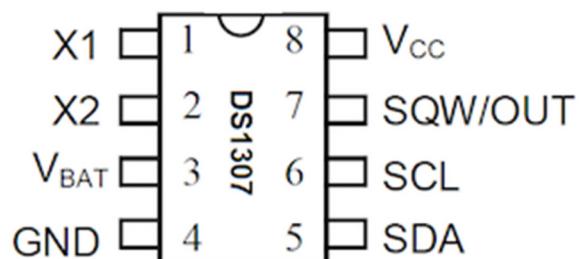
RTC Datasheet: <https://cdn.sparkfun.com/datasheets/BreakoutBoards/DS1307.pdf>

Theory

There are two unique parts to this project. The first is the four-digit seven-segment display. There are 12 pins on this device and 32 LEDs. To make this physically possible, eight of the pins are anodes, and four are cathodes. The cathodes control which digit is being grounded, while the anodes determine which segment of all the digits are being powered. This means that if you grounded all four digits and put power to one of the anode pins, four LEDs would come on. In order to achieve maximum capability with this display, the data that is sent to the anode pins must sync with one ground being turned on. If this is done for all four digits in a row, incredibly fast, the persistence of vision effect (POV) will come into play and it will appear as though all four digits are displaying different numbers simultaneously.



The next component is the real time clock (RTC) or specifically, the DS1307 chip. The goal of this real time clock is to, one flashed with the appropriate time, keep track of the number of milliseconds that have passed since January 1st, 1970. This Unix time allows the RTC to feed real time data to the microcontroller. The chip itself has eight pins, seven of which are essential.



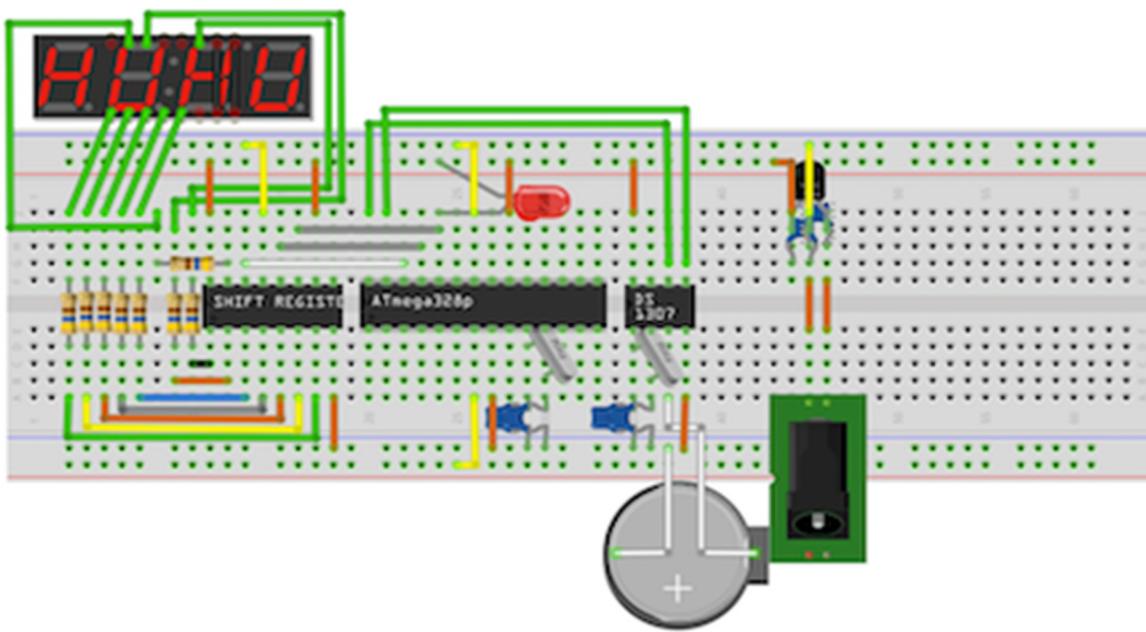
Firstly, there are rail and battery power pins that simply decide how the chip will be powered. Because of the onboard battery, the chip doesn't have to be flashed again if the rest of the breadboard circuit loses power. Of course, there is also a ground. Next, X1 and X2 are in charge of using a crystal in case the user wants an alternate clock speed. Finally, the Serial Clock Line (SCL) and the Serial Data Line (SDA) communicate with the microcontroller, with the data line sending the information and the clock line synchronizing the data transfers. To handle all the resistors, crystals, and capacitors for this chip, a board can be purchased. Once the RTC is fully installed, a library can be used to retrieve information from it.

Procedure

To start of this design, the microcontroller is hooked up to an 8MHz crystal, given power and wired to a programmer. Next, the anode pins of the display are wired through a 470Ω resistor network and into the output pins of a shift register. The cathode pins are connected directly to the off board ATmega328p microcontroller. For the final step of wiring the display, the clock, latch, and data pins of the shift register are also connected to the controller. Two normally closed push buttons (PBNC) are wired with pull down resistors and connected to the ATmega. These buttons take user input to either snooze the alarm or set the time.

Parts	Quantity
4 Digit 7 Segment Display	1
ATmega328p	1
Pocket AVR Programmer	1
Shift Register (SN74HC595)	1
Real Time Clock (DS1307)	1
5V Voltage Regulator (LM78L05)	1
470Ω Resistor Network	2
1KΩ Resistor	4
100uF Capacitor	6
Super Bright LED (YSL-R1042WC-D15)	1

Next, the two switches are wired to power, a pin on the microcontroller, and ground. These switches give the user control over when the clock is displaying the alarm and when they are setting the alarm. For the RTC, the SDA and SCL pins are hooked up to A4 and A5 on the microcontroller and it is given power and ground. A DC jack with a 5V voltage regulator is wired to allow the clock to run off a battery or an electrical plug. For the final component of the breadboard prototype, an LED is simply wired from an output pin of the ATmega to ground. Since this is the alarm of the circuit, the brighter the LED, the better. Once this product is all wired up, a case is 3D printed with holes for all the switches, buttons and LEDs. Code is written to POV the seven segments, take inputs from the buttons and switches, and to light up the LED when the alarm time is equal to the current time.



Code

```
// Name      : Light Based Alarm Clock
// Author    : Daniel Raymond
// Status    : Functional

// Global variable setup here
// Time Variables
volatile uint8_t alrmHr = 11;
volatile uint8_t alrmMin = 45;
// Switch & Button Variables
volatile uint8_t btnWork = 1; // See if button variable can be re-pressed

// Libraries
#include <TimerOne.h>
#include <Wire.h>
#include "RTClib.h"
#include <EEPROM.h>
RTC_DS1307 rtc;

// Pins
// Shift register pins
const uint8_t Data = A3;
const uint8_t Ltch = A2;
const uint8_t Clk = A1;
// All 7 segment ground pins
const uint8_t Seg1 = 4;
const uint8_t Seg2 = 5;
const uint8_t Seg3 = 6;
const uint8_t Seg4 = 7 ;
// Button pins
const uint8_t setBtn = 3;
const uint8_t offBtn = 2;
// Switch pins
const uint8_t setSwtch = 8;
const uint8_t alrmSwtch = 9;
// LED pins
const uint8_t LED = A0;
```

```
void setup() { // Runs once at the beginning
    if (! rtc.begin()) {
        while (1);
    }
    rtc.adjust(DateTime(F(__DATE__), F(__TIME__))); // Updates RTC

    DDRD = 0xF0;
    DDRC = 0x0F;
    // Allows button to be reactivated every 8th of a second
    Timer1.initialize(125000);
    Timer1.attachInterrupt(sec);
}

// Main Loop
void loop() {
    check();
    // Snoozes
    if (digitalRead(offBtn) == 0)
        digitalWrite(LED, LOW);
    // Checks switches for display
    if (digitalRead(alrmSwtch) == 0)
        tmDsplay();
    else {
        if (digitalRead(setSwtch) == 0)
            alrmDsplay();
        else
            setAlrm();
    }
}
// Displays the Time
void tmDsplay() {
    DateTime now = rtc.now();
    // Shows time from the RTC
    segment(Seg1, EEPROM.read(now.minute() % 10));
    segment(Seg2, EEPROM.read(now.minute() / 10));
    segment(Seg3, EEPROM.read(now.hour() % 10) | 1); // Turns decimal on
    if (now.hour() >= 10)
        segment(Seg4, EEPROM.read(now.hour() / 10));
}
// Displays what the alarm is
void alrmDsplay() {
    segment(Seg1, EEPROM.read(alrmMin % 10));
    segment(Seg2, EEPROM.read(alrmMin / 10));
    segment(Seg3, EEPROM.read(alrmHr % 10) | 1);
    if (alrmHr >= 10)
        segment(Seg4, EEPROM.read(alrmHr / 10));
}
```

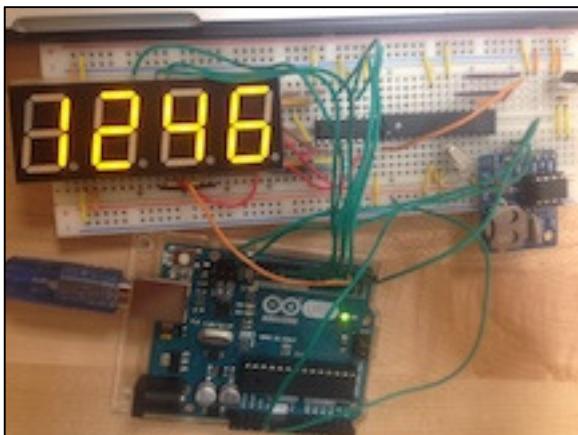
```

void setAlrm() { // Displays set mode for Alarm
    // Sets display
    for (int i = 0; i < 70; i++) {
        segment(Seg1, EEPROM.read(alrmMin % 10));
        segment(Seg2, EEPROM.read(alrmMin / 10));
        segment(Seg3, EEPROM.read(alrmHr % 10) | 1);
        if (alrmHr >= 10)
            segment(Seg4, EEPROM.read(alrmHr / 10));
    }
    if (digitalRead(setBtn) == 0 && btnWork == 1) { // Increases alarm
        alrmMin++;
        if (alrmMin == 60)
            alrmHr++;
        alrmMin %= 60;
        if (alrmHr == 25)
            alrmHr = 1;
        btnWork = 0;
    }
    blanked();
    delay(100);
}
void blanked() { // Blanks screen
    digitalWrite(Seg1, HIGH);
    digitalWrite(Seg2, HIGH);
    digitalWrite(Seg3, HIGH);
    digitalWrite(Seg4, HIGH);
}
void segment(uint8_t seg, uint8_t data) { // Shows data on one segment
    // Stops sending data
    digitalWrite(Ltch, LOW);
    shiftOut(Data, Clk, MSBFIRST, 0);
    digitalWrite(Ltch, HIGH);
    // Ground config
    blanked();
    digitalWrite(seg, LOW);
    // Data sending
    digitalWrite(Ltch, LOW);
    shiftOut(Data, Clk, MSBFIRST, data);
    digitalWrite(Ltch, HIGH);
    delay(1);
}
void sec() { // Re-allows button to increase variables every 1/8 second
    btnWork = 1;
}
void check() {
    DateTime now = rtc.now();
    // Checks if alarm should go off
    if (now.minute() == alrmMin && now.hour() == alrmHr && now.second() < 3)
        digitalWrite(LED, HIGH);
}

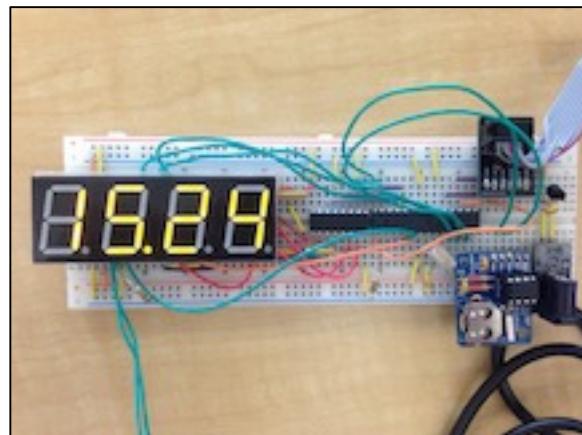
```

Media

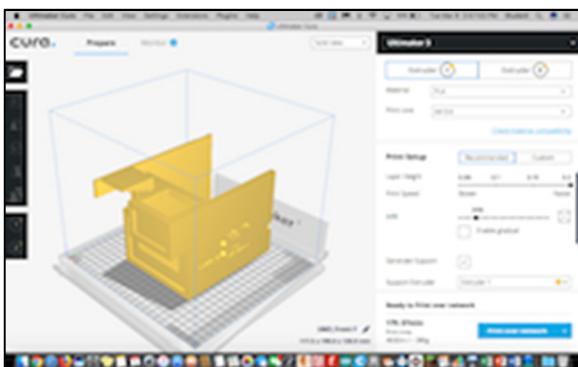
Personal Video: <https://youtu.be/sdNTqVRmALs>



Circuit Version 1 – Arduino



Circuit Version 2 – RTC + Off-board



ID Print Model



Final Product

Conclusion

I have never put more hours into a school project. I am very proud of the outcome, and love it show it off every chance I get. The fact that something I have made is something I want to use is fantastic. One thing that I love about ISPs is that whenever you decide your idea, you never have any idea of how to make it or how it will work. For instance, when I was first told about the RTC, I was very confused. Now, I feel as though I could wire one up in seconds (to be honest, I probably could). Overall, I am overjoyed by the whole experience.

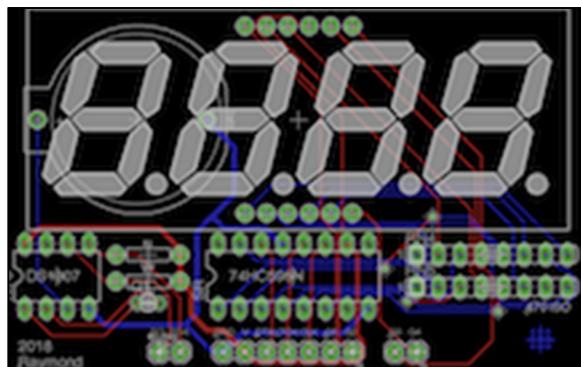
Project 13. Printed Circuit Board I: RTC Clock

Purpose

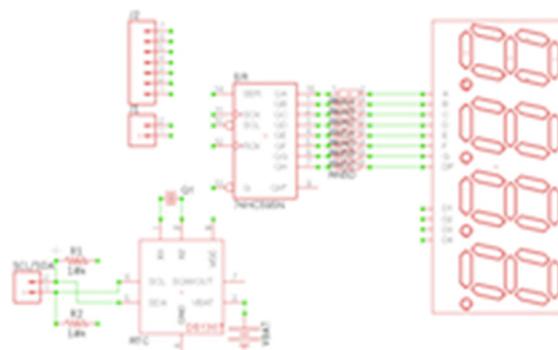
This project is an introduction to Eagle and designing printed circuit boards (PCB). For this, a board is designed with a four-digit seven segment display that is hooked up to a shift register. All the required control pins are wired to pins that can be hooked up to an Arduino. On top of this, a prewired RTC can be connected to the Arduino. With this project, a user can extract data from the RTC and then show it on the display.

Parts	Quantity
Printed Circuit Board	1
4 Digit 7 Segment Display	1
Real Time Clock (DS1307)	1
8MHz Crystal	1
Shift Register (SN74HC595)	1
470Ω Resistor Network	2
10kΩ Resistor	2

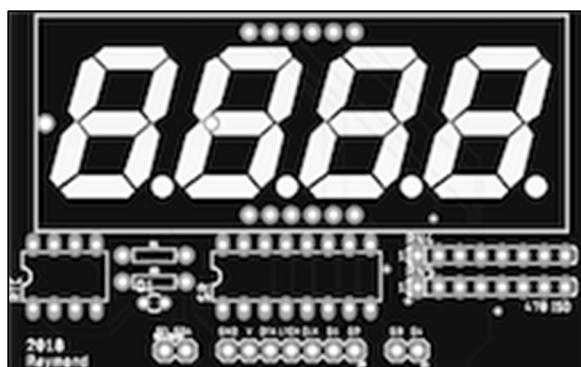
Media



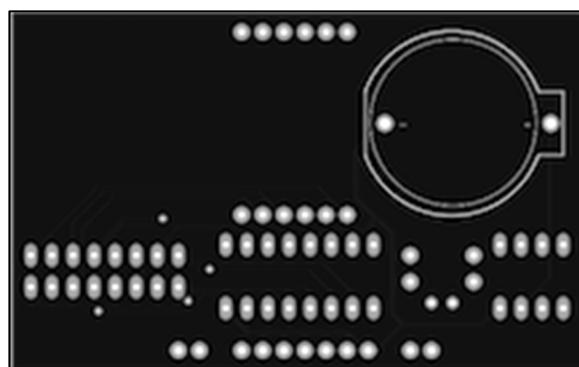
Finished Eagle Design



Schematic



Rendered Top



Rendered Bottom

Project 14. Printed Circuit Board II: RTC Clock

Purpose

The purpose of this project is to complete the PCB design project by soldering, testing, and coding. This product is designed for grade 11 ACES who are learning how to use real time clocks (RTC), specifically the DS1307. The quad 7 segment display gives the user a finished product, as well as another component to use alongside the RTC.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html#PCB2>

RTC Datasheet: <https://cdn.sparkfun.com/datasheets/BreakoutBoards/DS1307.pdf>

DirtyPCBs: <https://dirtypcbs.com/store/pcbs>

Theory

The theory for this PCB is quite simple. Like my previous ISP, the Light-Based Alarm Clock, a microcontroller reads a DS1307 RTC and sends the data directly to a quad 7 segment display. Since this RTC is I2C compatible device, only four wires are needed to communicate with it: SCL, SDA, power, and ground. Since there are four digits on the display and only two numbers, hours and minutes, modulo and integer division are used to separate the high digit from the low one. Once the microcontroller has read in the time from the flashed RTC, it sends the data to the display using an eight-bit serial in parallel out shift register. Since this specific display only has 12 pins, the four grounds for each of the 7 segments are used to show all the digits using POV.

Procedure

After designing the PCB in eagle, the layers of the board are separated using a CAM processor and sent to a PCB manufacturer, in this case DirtyPCBs. Once the boards arrive two chip seats are soldered on; one for the RTC and one for the shift register. The two $10\text{K}\Omega$ resistors, resistor networks, capacitor, and battery holder are soldered in as well. Finally, the display and headers pins are soldered in. The RTC and shift register are inserted into the chip seats. To connect the board to the Arduino, the PCB is inserted on the digital pin side of the board with ‘G4’ being inserted into digital pin six. Finally, code is written to read the data of the RTC and deliver it to the display using POV.

Parts	Quantity
PCB	1
4 Digit 7 Segment Display	1
Arduino UNO	1
Shift Register	1
Real Time Clock (DS1307)	1
5V Cell Battery & Holder	1
Chip Seat	2
470 Ω Resistor Network	2
10K Ω Resistor	2
100uF Capacitor	6
Male Header Pins	11

Code

```
// Name    : PCB RTC Clock
// Author  : Daniel Raymond
// Status   : Functional

// abcdefg.

#include <EEPROM.h>
#include <TimerOne.h>
#include <Wire.h>
#include "RTClib.h"

RTC_DS1307 rtc;
const uint8_t Pwr = 13;
// Shift register pins
const uint8_t Data = 12;
const uint8_t Ltch = 11;
const uint8_t Clk = 10;
// All 7 segment ground pins
const uint8_t Seg4 = 9;
const uint8_t Seg3 = 8;
const uint8_t Seg2 = 7;
const uint8_t Seg1 = 6 ;

void setup () {
  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);

  if (! rtc.begin())
    while (1);
  if (! rtc.isrunning())
    rtc.adjust(DateTime(F(__DATE__)), F(__TIME__)));

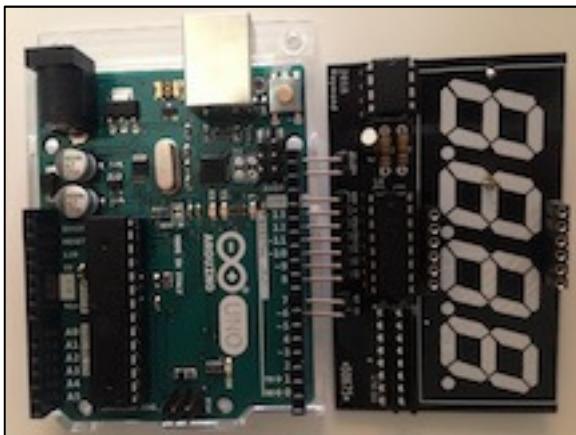
  for (int i = 6; i < 14; i++)
    pinMode(i, OUTPUT);
  digitalWrite(Pwr, HIGH);
}

// Main Loop
void loop() {
  DateTime now = rtc.now();
  // Shows time from the RTC
  segment(Seg1, EEPROM.read(now.minute() % 10));
  segment(Seg2, EEPROM.read(now.minute() / 10));
  segment(Seg3, EEPROM.read(now.hour() % 10));
  // // if (now.hour() >= 10)
  segment(Seg4, EEPROM.read(now.hour() / 10));
}
```

```
// Shows data on one segment
void segment(uint8_t seg, uint8_t data) {
    // Stops sending data
    digitalWrite(Ltch, LOW);
    shiftOut(Data, Clk, LSBFIRST, 0);
    digitalWrite(Ltch, HIGH);
    // Ground config
    digitalWrite(Seg1, HIGH);
    digitalWrite(Seg2, HIGH);
    digitalWrite(Seg3, HIGH);
    digitalWrite(Seg4, HIGH);
    digitalWrite(seg, LOW);
    // Data sending
    digitalWrite(Ltch, LOW);
    shiftOut(Data, Clk, LSBFIRST, data);
    digitalWrite(Ltch, HIGH);
    delay(1);
}
```

Media

Personal Video: <https://youtu.be/DA6a45qH49Y>



PCB Pin layout



Clock in Action

Conclusion

Receiving a PCB that I had made for their first time in my life was exhilarating. Seeing physical boards that were very professional made me see the potential for future projects and ISPs. Soldering the board was nerve-racking, as I thought there was no way it was ever going to work first try; to my surprise, it did. I was so excited to solder it that I forgot to take video of the process! After that, coding it was a breeze because I had done a project like this for my ISP. Overall, I am thrilled with how my design turned out; I love its compact design and professional style.

Project 15. I2C Velocity Data Logger

Purpose

The goal of this project is to develop an understanding of the Inter-Integrated Circuit (I2C) protocol. This is achieved by creating a circuit which logs data onto an external electrically erasable programmable read-only memory (EEPROM) chip. This data is then presented on the serial plotter by reading off said chip. An RTC is also used to time stamp each byte of data. The data that is going to be logged is the velocity of a hand that is using a computer.

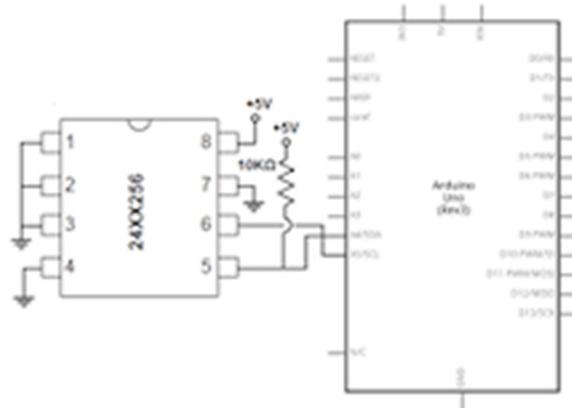
Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/Tasks.html>

EEPROM Datasheet: http://www.advanide.com/wp-content/uploads/products/contact_memories/24LC256.pdf

Theory

To calculate the velocity of the hand, first the coordinates of it are measured. These values are then subtracted from its previously measured coordinates to figure out the total distance travelled over a given period. From these two lengths, x and y, Pythagorean theorem is used to calculate the total distance the object travelled. This distance value is divided by the period and then logged onto an external I2C EEPROM chip. EEPROM chips have what is essentially an array of bytes, that can be read and written to. To measure the period in j distance recordings, an RTC is used. The RTC has a square wave, which emits a 1Hz, 4KHz, 8kHz, or 32KHz waves. This allows frequency allows us to determine how often we sample from our sensors.



EEPROM Wiring

Procedure

A DS1307 RTC chip is connected to the SDA and SCL pins of an Arduino UNO. Pull up resistors are tied to both lines, as the Arduino can only pull them down. A 24LC256 EEPROM chip is also wired as shown in *EEPROM Wiring* so that the microcontroller can read and write to it. The sensors that are used for this project are two IR distance sensors.

This allows for two-dimensional distance tracking of an object. Using this information, the velocity can be calculated. This is the data that is sent to the EEPROM to be stored. Finally, the data is read off the EEPROM chip and displayed on a serial plotter.

Parts	Quantity
Arduino UNO	1
Real Time Clock (DS1307)	1
I2C EEPROM (24LC256)	1
IR Distance Sensor (GP2Y0A41SK0F)	2
10KΩ Resistor	2

Code

```
// Name    : I2C Speed Graphing Program
// Author  : Daniel Raymond
// Status   : Functional

#include <Wire.h>
#define RTC_ADDRESS 0x68
#define EEPROM_ADDRESS 0x50 // Stores 32KB
#define INT0 2
#define SQWE 4
#define controlReg 0x07
#define RS0 0
#define RS1 1
#define Hz4 (1 << RS0)
#define smth 3 // Smoothing factor

uint16_t averaging = 0;
uint8_t incr = 0;
volatile uint16_t count = 0;
volatile bool triggered = false;

double distance = 0;
double tmDif = 0;
double avVels[smth];
double avVel = 0;

struct prevRec {
    double seconds;
    double minutes;
    double distX;
    double distY;
    double mills;
} pr;
struct currRec {
    double seconds;
    double minutes;
    double distX;
    double distY;
    double mills;
} cr;

void setup() {
    Serial.begin(9600);
    Wire.begin();
    Wire.beginTransmission(RTC_ADDRESS);
    Wire.write(controlReg);
    Wire.write((1 << SQWE) | Hz4); // Sets the Hz at which we receive data
    Wire.endTransmission();

    PORTD |= (1 << PD2); // Enable pullup resistor (Input Mode + writing HIGH)
    attachInterrupt(digitalPinToInterrupt(INT0), pulse, FALLING);
}
```

```

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

    Wire.beginTransmission(RTC_ADDRESS);
    Wire.write(controlReg);
    Wire.write((1 << SQWE) | Hz4); // Sets the Hz at which we receive data
    Wire.endTransmission();

    PORTD |= (1 << PD2); // Enable pullup resistor (Input Mode + writing HIGH)

    attachInterrupt(digitalPinToInterrupt(INT0), pulse, FALLING);
}

void loop() {
    if (triggered) { // Runs times a second
        triggered = false;

        avVels[incr] = calcVel(); // Smooth out the data
        incr = (incr + 1) % smth;

        avVel += avVel[incr];

        if (incr == (smth-1)) {
            Serial.println((avVel/smth));
            avVel = 0;
        }
    }
} // End loop
void readDist() {
    uint8_t rawX = analogRead(PC0);
    uint8_t rawY = analogRead(PC1);

    pr.distX = cr.distX;
    pr.distY = cr.distY;

    cr.distX = constrain(2076 / (rawX - 11), 4, 30);
    cr.distY = constrain(2076 / (rawY - 11), 4, 30);

    double lengthX = abs(cr.distX - pr.distX);
    double lengthY = abs(cr.distY - pr.distY);

    distance = sqrt(lengthX * lengthX + lengthY * lengthY);
}

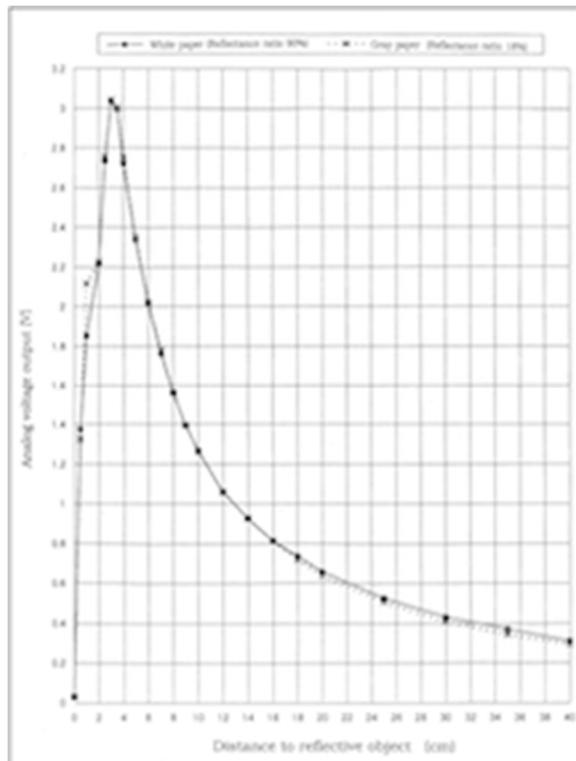
double calcVel() {
    readDist();
    pr.mills = cr.mills;
    pr.seconds = cr.seconds;
    pr.minutes = cr.minutes;
    Wire.beginTransmission(RTC_ADDRESS);
    Wire.write(0);
    Wire.endTransmission();
    Wire.requestFrom(RTC_ADDRESS, 2);
    while (!Wire.available());
    cr.seconds = bcd2dec(Wire.read());
    cr.minutes = bcd2dec(Wire.read());
    cr.mills = count;
    tmDif = ((cr.minutes - pr.minutes) * 60) + (cr.seconds - pr.seconds) +
    ((cr.mills - pr.mills) / 4096);
    return (distance / tmDif) / 100;
}

```

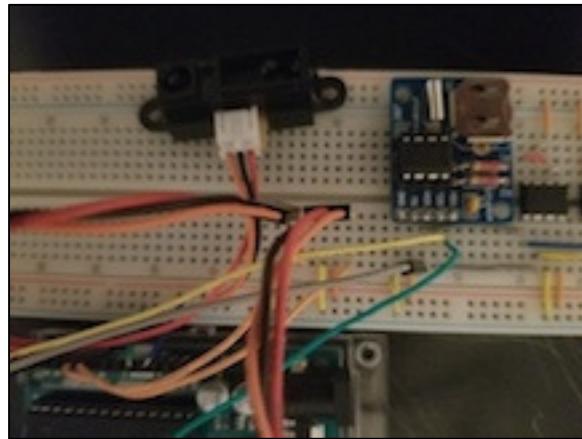
```
void pulse() {  
    count++;  
    triggered = true;  
    PORTB ^= (1 << PB5); // Flicker the LED (Alternates between 0 & 1)  
}  
  
uint8_t bcd2dec(uint8_t bcd) {  
    return 10 * (bcd >> 4) + (bcd & 0x0F);  
}
```

Media

Personal Video: <https://youtu.be/IMiuRwL740Q>



Distance Sensor Distance to Voltage Graph



Circuit

Conclusion

Although it was disappointing not to get the EEPROM chip working, I still learned a lot about I2C. I liked the idea of incorporating physics concept into hardware, because it feels as though this project has a real-world application. Once again, for the last time, I've realized how inconsistent the distance sensors are. Although this project was late, rushed, and didn't work, I still feel as though I put way too many hours into this project, especially with this ISP coming up.

Project 16. Medium ISP: Bluetooth Photo Frame

Purpose

The purpose of this project was to design and create a device that would receive photos using Bluetooth and display them on a graphical liquid crystal display (GLCD). The photos would be sent from a phone that has a personally designed app to transfer the data.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI3M/1718/ISPs.html>

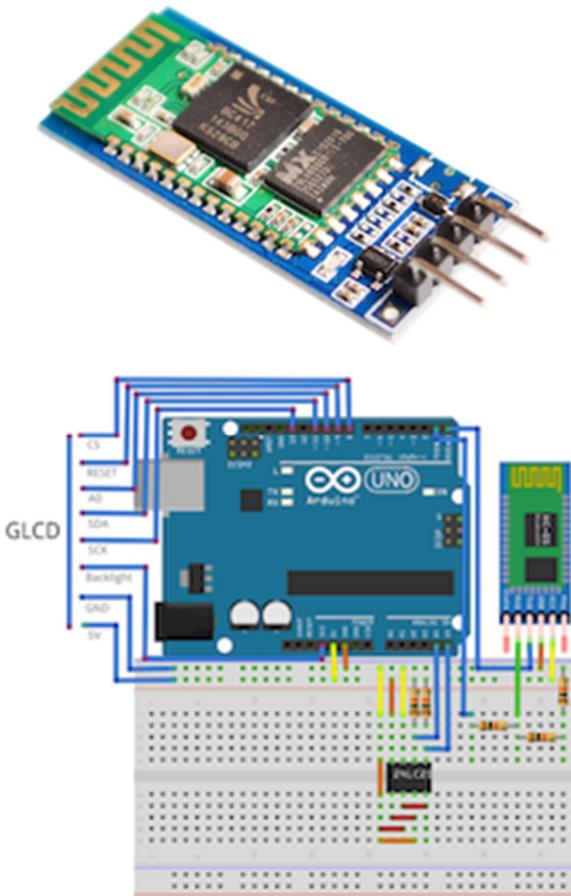
GLCD Datasheet: <https://www.buydisplay.com/download/manual/ER-TFTM050-2>

MIT App Inventor: <http://ai2.appinventor.mit.edu/#>

Theory

The HC05 module itself handles most of the difficult aspects of Bluetooth. It transmits data via low power radio wave frequencies, specifically around 2.45 gigahertz (GHz). This frequency is an internationally agreed upon standard. Since Bluetooth data is transferred through radio waves, this means that it must either use amplitude modulation (AM) or frequency modulation (FM). As it turns out, Bluetooth uses FM. If the phone wants to send a 0 to the module, it sends out a frequency of 2.402 GHz, and if it wants to send out a one, it sends out 2.480 GHz. Bluetooth wiring and coding, however, is quite simple. There are two communication pins for transmitting and receiving data. The transmitter of the HC05 is connected to the receiver of the Arduino, and the reverse is true for the other communication pins. Since this is serial communication, the Arduino pins used are zero and one. Whenever, the Arduino receives data from the outside world, all the code must do is state that when the serial in line has something on it, read it.

The GLCD has three main components: vertical sync (vsync), horizontal sync (hsync), and colour. The hsync chooses the column in which the colour is going to be displayed on. To do this, it scrolls along the screen horizontally. The vsync is coordinating with it in such a way that, whenever the hsync finishes scrolling through a row, the vsync shifts down by one. This creates a unique point in time for each pixel to be told what colour to be. All the microcontroller has to do is tell the GLCD what colour to display at any given point in time.



The phone application development software is very high level. It uses block code and along with drag-and-drop interfacing. All the app does is connect with the HC05 and have a button to tell it which image to display. When the button is pushed, it sends the appropriate data.

Parts	Quantity
Arduino UNO	1
HC05 Bluetooth Module	1
I2C EEPROM (24LC256)	1
ILI9163C Graphic LCD	1
4.7KΩ Resistor	5

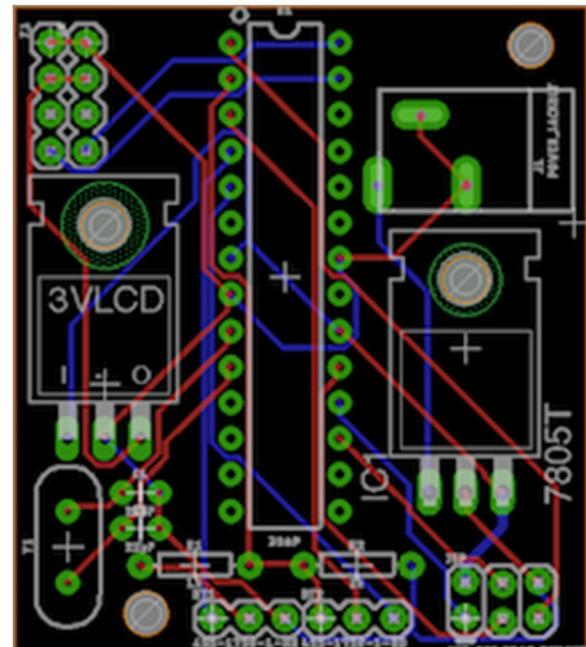
Procedure

To start off the project, the RX and TX pins of the HC05 Bluetooth module are connected to the TX and RX pins of the Arduino (respectively). This allows the Arduino to transmit data using Bluetooth, as well as receive data from the transmitting HC05. Since the logic level of the Bluetooth module is 3.3V, a voltage divider is used on the RX pin. 5V and ground are also connected to the HC05.

When the ER-TFTM0D0-2 is purchased, the decision must be made of what kind of communication to use. Since the class is most familiar with it, I²C is chosen. This means that the SDA and SCL pins need to be connected, as well as power and two ground. This is a beast of a machine due to its onboard microcontroller, so it requires 300mA of current. For this, a PCB is designed to provide enough power to the GLCD, as well as an offboard ATmega328p microcontroller to run it. This specific display does not have a backlight pin, so registers need to be manipulated in order to turn it on. Unfortunately, the difficulty of this display was underestimated, so a new display is purchased that is much, much simpler. The new display is the ILI9163C graphic LCD. It uses SPI communication on top of having a dedicated power, ground, and backlight pin.

After the entire device is wired up and programmed, an app is developed for Android that connects to the HC05, selects an image, and sends it to the Arduino. Unfortunately, there is no feasible way to transfer image data over Bluetooth; it can only send and receive text. This means that the data transfers would have to be simple. To work around this distinct problem, external EEPROM chips are used to store pre-set images. When the app sends a string, the Arduino receives it and determines which image is to be shown on the GLCD screen. The specific EEPROM chip that is the 24LC256 I²C EEPROM chip, which have been used (unsuccessfully) in a previous project. Once the EEPROM is in place, the prototype is finally complete.

If one were to go a step further in this process, a new PCB would be designed to accommodate the new parts and a case would be 3D printed to store the project. Unfortunately, time constraints did not allow for these extremities.



PCB for the ER-TFTM0D0-2

Code

```
// Name: Bluetooth Photo Frame
// Author: Daniel Raymond
// Description: Receives photos using Bluetooth and then displays them on a GLCD.

#include <Wire.h>
#include <SPI.h>
#include <Adafruit_GFX.h>
#include <TFT_ILI9163C.h>

#define EEPROM_ADDRESS 0x50
#define CSelect 10
#define DC 9
#define a0 8
#define BLACK 0x00
#define baud 9600
#define LCD_W 128
#define LCD_H 128
uint8_t state = 0;

TFT_ILI9163C tft = TFT_ILI9163C(CSelect, a0, DC);

void setup() {
    tft.fillScreen(BLACK); // Start off with a black screen
    Wire.begin(); // Begin wire library
    tft.begin(); // Begin tft library
    Serial.begin(baud); // Default communication rate of the Bluetooth module
}

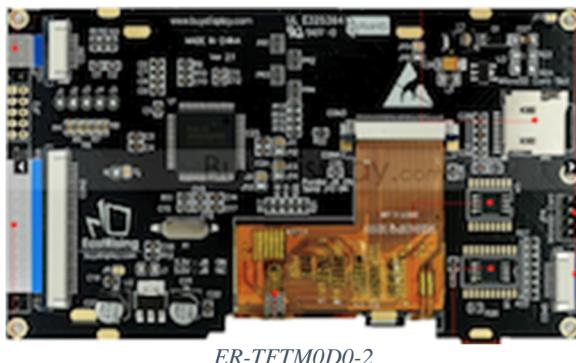
void loop() {
    if (Serial.available() > 0) { // Checks whether data is coming from the serial port
        state = Serial.read(); // Reads the data from the serial port
        for (uint8_t x = 0; x < LCD_W; x++)
            for (uint8_t y = 0; y < LCD_H; y++) // Read colour data from EEPROM
                tft.drawPixel(x, y, convClr(readIIC(EEPROM_ADDRESS, (x*12+y)+(state-48)*16834)));
        // state is a character: subtracting 48 converts to an int and multiplies by
        // 16834 to determine which image
        state = 0; // Reset state
    }
}

uint16_t convClr(uint8_t clr) {
    uint8_t red = clr >> 5; // 3 bits
    uint8_t green = (clr >> 2) & B00111; // 3 bits
    uint8_t blue = clr & B11; // 2 bits
    return (map(red, 0, 8, 0, 0xFF) << 8) | (map(green, 0, 8, 0, 0xFF) << 4) | map(blue, 0, 4, 0, 0xFF); // Maps 8bpp to 16bpp
}

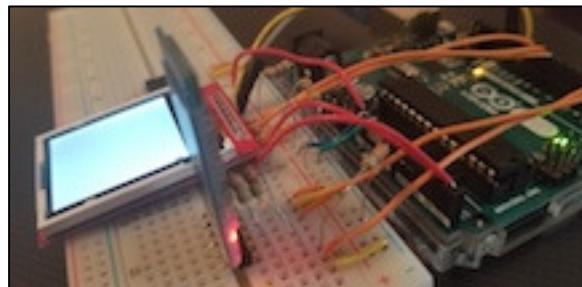
uint8_t readIIC(int adrs, unsigned int eeaddress) {
    uint8_t rdata = 0xFF; // sets a uint8_t for data
    Wire.beginTransmission(adrs); // begins communicating with eeprom address
    Wire.write((int)(eeaddress >> 8)); // sends the MSB to the eeprom address
    Wire.write((int)(eeaddress & 0xFF)); // sends the LSB to the eeprom address
    Wire.endTransmission(); // end transmission
    Wire.requestFrom(adrs, 1); // request one uint8_t from that address
    while (!Wire.available());
    rdata = Wire.read(); // if the data is available read it
    return rdata; // return the data at specified address
}
```

Media

Personal Video: <https://youtu.be/twDLak1YYAc>



ER-TFTM0D0-2



Display Prototype

```
when [ListPicker1].BeforePicking
do set ListPicker1. Elements to BluetoothClient1. AddressesAndNames

when [ListPicker1].AfterPicking
do set ListPicker1. Selection to call BluetoothClient1. Connect
address ListPicker1. Selection
set ListPicker1. Text to Obfuscated Text "Connected"
set ListPicker1. Height to ListPicker1. Text

when [Button1].Click
do call BluetoothClient1. SendText
text 1

when [Button2].Click
do call BluetoothClient1. SendText
text 0
```

MIT App Inventor 2 Block Code

Conclusion

I underestimated the difficulty of this project. The Bluetooth is finicky, the graphic LCDs are infinitely more complex than LED matrices, and phone applications are limiting. This is the first time an ISP of mine has not gone to plan in any sort of capacity. Failure is something that, while uncomfortable at first, has shown me to appreciate the knowledge gained more than the final project. I fully believe that I will be using these parts in the future, and the skills I have gained from this ISP will most likely help me in my future endeavors.

ICS4U: AVR Optimization

Project 17. CHUMP

Part A. The Code

Purpose

This project is an introduction to machine code, assembly, and the future project of the four-bit computer. The assignment is to come up with a series of machine code commands that performs a task. Since this project is only based on the code, the mechanics of the four-bit computer will be explained in a later report.

Theory

There are seven general commands, all of which have two flavors: memory and constant. If the constant version is chosen, the lower nibble acts as the number in which the command uses. If the memory version is chosen, it acts as the address for the register in RAM. The data in this register is the number that the command uses. All the commands alter data in registers; it is their function. Figure 1 demonstrates each command and what register they change.

LOAD	.Accumulator
ADD	.Accumulator
SUBTRACT	.Accumulator
STORETORAM
READAddress
GOTOPC
IFZEROPC

Figure 1

The goal of this specific series of commands is to check if the accumulator register is equivalent to the second register (register 1) in random access memory (RAM). To do this, it first sets register 1 to a random value; in this case, two. Then it loads the accumulator with the value it wants it to be compared to. Once all of this preparation is done, the computer subtracts three (the value from register 1) from the accumulator. If this is then equal to zero, the two numbers are equivalent. To signify this equivalency, the computer places a one in register five. If they are not equivalent, it leaves it at zero. Finally, it resets the accumulator back to its original value of two.

Code

0000:	00000010	LOAD	2	; accum = 2
0001:	10000001	READ	1	; addr = 1
0002:	01100001	STORETO	1	; memory[1] = 2
0003:	00000011	LOAD	3	; accum = 3
0004:	01010001	SUBTRACT	2	; accum -= mem[1]
0005:	10101000	IFZERO	8	; accum == 0? pc = 8
0006:	00000001	LOAD	1	; accum = 1
0007:	01110101	STORETO	5	; mem[5] = 1
0008:	00000001	LOAD	2	; accum = 2

Reflection

This project was complex and mind-bending. Although all the registers, loading, and arithmetic was challenging itself, I found the most difficult part about this entire project was finding an end goal with a level of appropriate difficulty. The processor was so limited that I didn't fully understand what purpose it served, and thus it took me a while to come up with broken code.

Part B. The Clock

Purpose

The clock is the heartbeat of a computer. It synchronizes all the data transmission and coordinates components of a computer. In this report, multiple timer ICs will be used to create this clock.

Reference

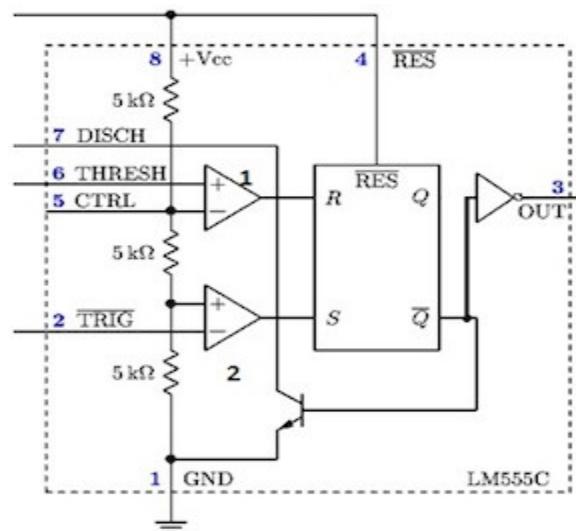
Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#tasks>

555 Explained: <https://youtu.be/kRISFm519Bo?t=2m59s>

Logic Gate Combination: <https://youtu.be/SmQ5K7UQPMM?t=45s>

Theory

This circuit has four main components: astable, monostable, and bistable oscillators, as well as a few logic gates that connects everything. All three oscillators use the 555 timer as their core IC. The 555 has a threshold and a trigger voltage that turn the output off and on respectively. When current is applied to pins six and two, comparators check to see if it is above 3.3V or below 1.7. If it is above, it will reset the SR Latch (See diagram to the right) and if it is below the chip will set the output to on. When the output is high, pin seven of the 555 is connected to ground. This is meant to drain the external capacitor, cycling the process. The 555 can be repurposed in many ways by changing the external components. It can even be used for its SR latch alone.

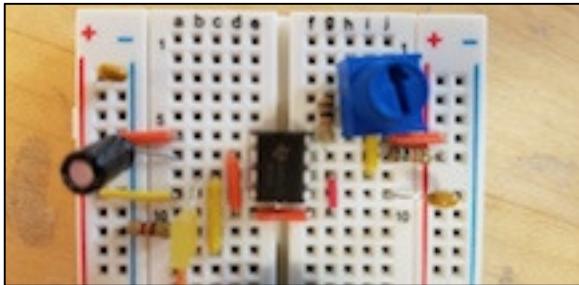


Procedure

First, an isolated astable oscillator is built around one of the 555 timer ICs. This circuit is designed to have a defined pulsing output with its duty cycle controlled by a potentiometer. Next, the monostable oscillator is built. This is not so much an oscillator as it is a debounced button using a 555. Then the bistable oscillator is built using a switch and yet another 555 IC. This is practically a switch that turns an output on and off; the 555 is again for debouncing. Finally, to combine all three oscillators, a switch along with a series of ‘and’ and ‘or’ gates are used to determine the final output: an automatic or manual pulse.

Parts	Quantity
555 Timer IC	3
SN74LS00N	1
SN74LS04N	1
SN74LS08N	1
1K Ω Resistor	6
Toggle Switch	2
PBNO Button	1
100nF Capacitor	4
1uF	1
3mm LED	3

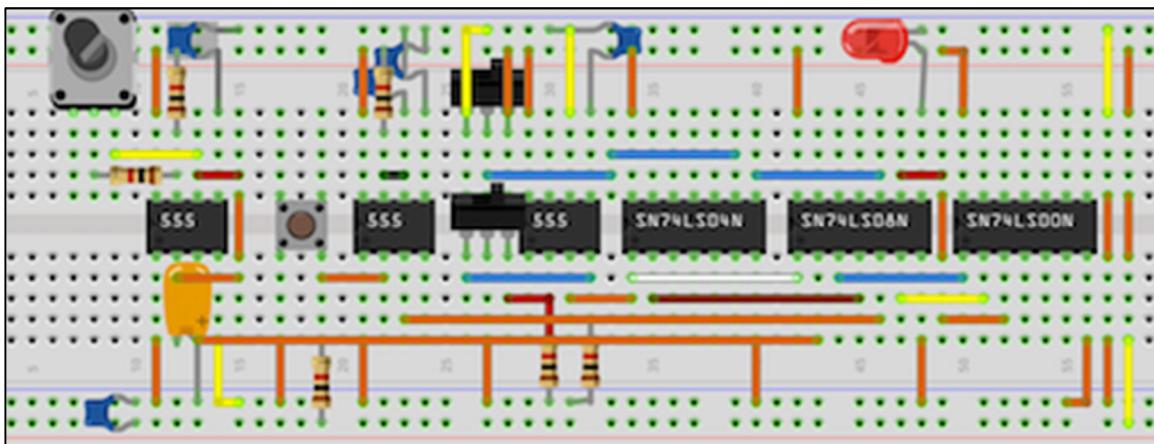
Media



Astable Oscillator



Bimetric view



Final Circuit

Reflection

This project was satisfying, partially because I finished it without too many hitches, but also because it seems like such a simple idea. If I showed the outcome of the LED (perhaps if this was a product of some kind), they wouldn't be impressed in the slightest. I mean, we made an LED blink, and switch to a button activated LED. Congratulations. Now, this may seem like a disappointment, but it is because of hardware that I have gained an appreciation for just how complex these things can be. Resistors and potentiometers all affect the outcome, one wrong wire and the entire thing is dysfunctional. Overall, I'd say this is the coolest way to run the blink sketch so far.

Part C. ALU: The Arithmetic and Logic Unit

Purpose

This project is all about learning how to use an arithmetic logic unit (ALU). Specifically, the purpose of this project is to learn how the select pins and other inputs are used to manipulate the functions of the ALU and thus the output.

Reference

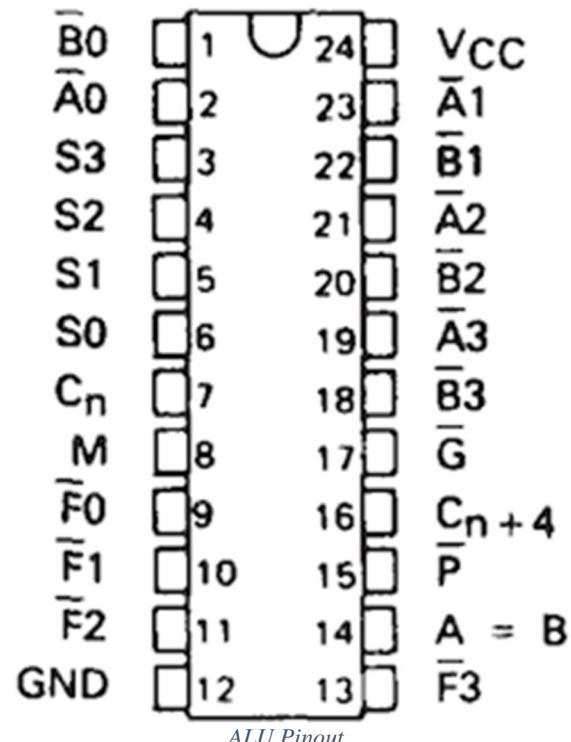
Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#ALU>

ALU Datasheet: <https://source.z2data.com/2016/10/12/6/26/577/639651/sn54ls181.pdf>

Theory

There are five main categories of pins that are used: select, carry, ‘A’, ‘B’, and output. On top of this there is a mode pin as well. The mode pin changes whether the ALU is performing logical or arithmetic operations. Since there are 16 logical and 32 arithmetic, the C_n pin is used to create the addition 16 arithmetic operations. C_{n+4} is the carry pin and is used to chain these processors together, creating a more powerful eight-bit ALU. Finally, there is the pin ‘A’ = ‘B’ which sets the constants used in the function equal.

This ALU is a four-bit processor. Because of this, all the remaining categories are split into groups of four pins. The select pins for example, determine which function to use. Since mode and C_n decide which group of 16 function to choose from, the select pins finish the job and choose the specific function to perform.



‘A’ and ‘B’ pins make up two separate constants that are manipulated by a function to create the final output. Since there are four pins for each constant, the numbers they can create are limited from zero to 15. ‘A0’ and ‘B0’ control the least significant bit of the nibble that makes up the constants, and ‘A3’ and ‘B3’ make up the most significant.

The final category of pins that are used in the core assembly of this ALU are the output pins. There are four pins to produce a four-bit output. Again, zero to 15. If the output from a given function was 13, then the ‘F3’, ‘F2’, ‘F1’, and ‘F0’ would show high, high, low, and high. It is important to note that there is no overflow with a single ALU; it will simply cut off the output at a maximum value of 15.

Procedure

To begin wiring the ALU for testing purposes, pins 24 and 12 are connected to voltage and ground respectively. Pin 14 (see ALU Pinout) is wired to ground as the constants should be separate.

The select pins (S_0-S_3) are attached to jumper wires. This allows for the user to have flexibility and easy access when testing the functions of the ALU. The same goes for the Mode and C_n pins as they also affect the type of functions that are used. Jumper wires are again used for ‘A’ and ‘B’ pins to easily change the constants used in the functions. To simplify the use of the circuit, the rail connections of select, ‘A’, and ‘B’ are separated. They are also colour-coded so that each category of pins can be easily identified.

The output pins (F_0-F_3) are wired to the anode pins of an LED bar graph. A 220Ω bussed resistor network is connected to each pin used in the bar graph, with its common pin attached to ground.

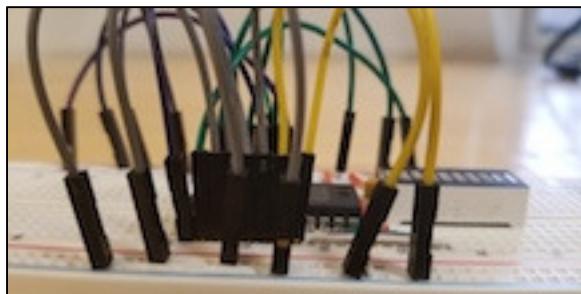
Five volts are applied to the circuit. To test it, the select pins are maneuvered to match the function table. The ‘A’ and ‘B’ pins are moved from power to ground and back again to ensure that the entire circuit is functional.

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

Functions Table

Media

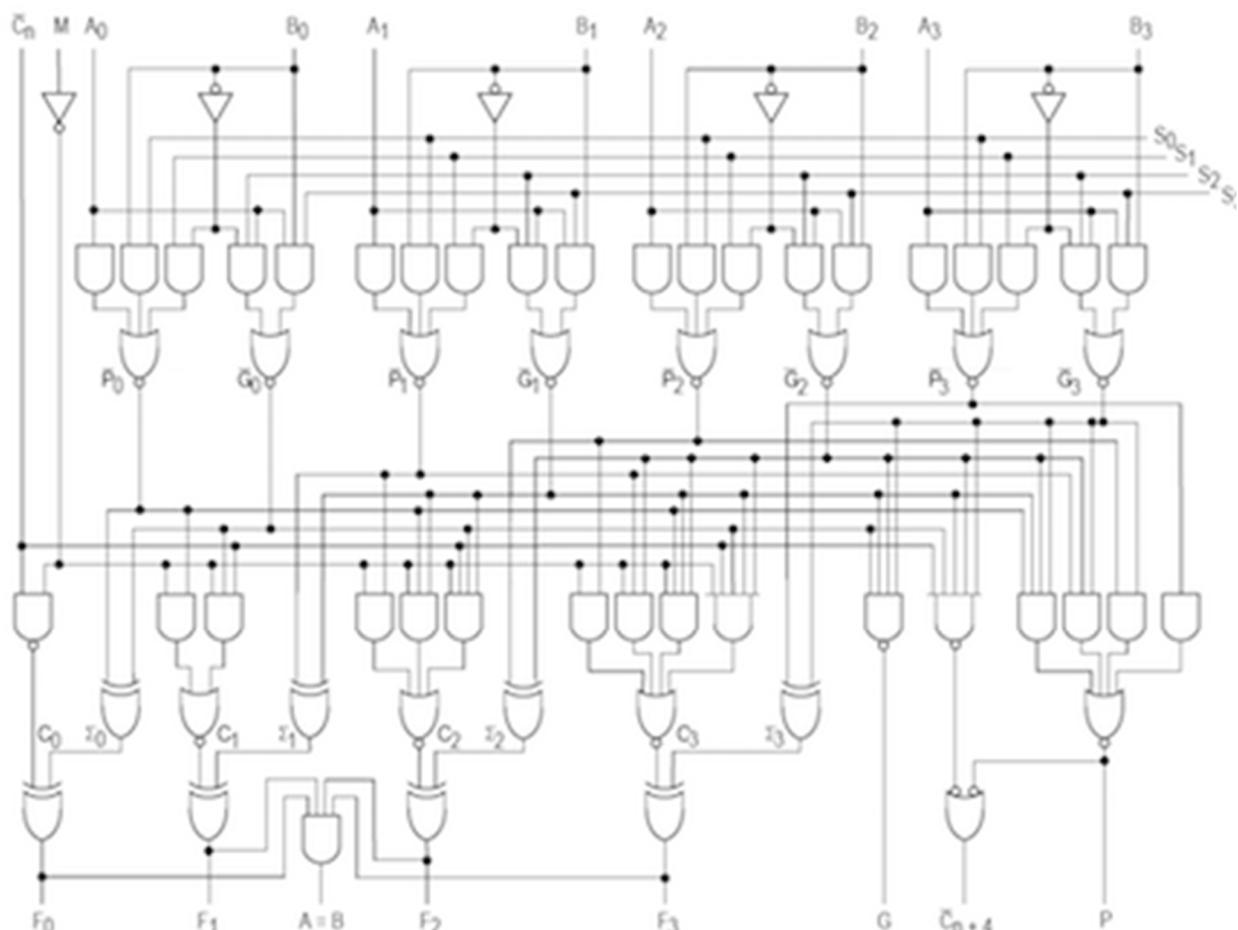
Personal Video: <https://youtu.be/wKXTI73b1jg>



Mode, C_n , and Select Pins



ALU – Trimetric View



SN74LS181N Schematic

Reflection

This project was not as difficult as I thought it would be. The ALU itself is relatively simple to understand if you have the correct table and data to look at. However, while this ALU is easy to understand, I feel that we will find it much more difficult to use the logic gates to perform an actual task.

Part D. The ROMs

Purpose

This project is the initial wiring of the complete CHUMP. In it the clock, program counter (PC), program EEPROM, and the input to the control EEPROM are wired together to create an LED output that displays the relevant data.

Reference

Home Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#Processor>

PC Datasheet: <http://mail.rsgc.on.ca/~cdarcy/Datasheets/74LS161.pdf>

ROM Datasheet: <http://mail.rsgc.on.ca/~cdarcy/Datasheets/AT28C17.pdf>

Theory

The program counter is simply a four-bit counter with an option to load another number into it. It receives a square wave and for each pulse it adds one to its output. This chip uses its four output pins to send a four-bit number, Q_A being the LSB and Q_D being the MSB. Its load pin is active low, so whenever it is grounded, the PC's register changes to match four input pins: A, B, C, & D (see [Program Counter Pinout](#)). Clear is an active low pin, and it resets the counter register.

The remaining three pins, ENT, ENP, and RCO, are for chaining these PCs together. RCO is an output pin that goes high only when its PC is at nine. This is then fed into the ENT of the next PC, creating a 2-digit binary coded decimal. However, this is not used for this project.

The next chip in the circuit is the AT28C17 EEPROM chip (ROM). It has 2048 eight-bit registers. To access each individual register, 11

address pins are used: A_0-A_{10} . This chip has two main modes: reading and writing. When writing is enabled, the output pins become input pins and change the register at the address specified by A_0-A_{10} . If output is enabled, these pins become the output that displays the data inside the addressed register and can be read. The only other unique pin on this chip is pin 1. It is an output pin that tells the reader whether this chip is ready to be used, or busy performing an internal operation.

RDY/BUSY	1	28	VCC
NC	2	27	WE
A7	3	26	NC
A6	4	25	A8
A5	5	24	A9
A4	6	23	NC
A3	7	22	OE
A2	8	21	A10
A1	9	20	CE
A0	10	19	I/O7
I/O0	11	18	I/O6
I/O1	12	17	I/O5
I/O2	13	16	I/O4
GND	14	15	I/O3

EEPROM Pinout

Procedure

To begin wiring this computer, power and ground are connected to all the rails and all the chips. The output of the [clock](#) is connected to the clock pin of the PC. This clock, in the future, will be synchronizing multiple components of the CHUMP, but for now it only connects to the PC. The load pin is connected to power as it is an active low pin. The ALU will be connected to the input pins of the PC in the future, but for now they remain unconnected.

Parts	Quantity
74LS161 (Program Counter)	1
AT28C17 (ROM)	2
Square wave source (Clock)	1
LEDs (R/Y/G)	4/9/8

The four output pins (Q_A - Q_D) are wired to the first four address pins of the program ROM with A_0 being the LSB. LEDs are connected for debugging and demonstrative purposes. Since the PC can only count to fifteen, the rest of the address pins cannot be controlled within the computer, so they are connected to ground. However, it is possible to access the higher addresses by connecting optional jumper wires to the remaining address pins instead of grounding them, leaving manual control to the operator. To make the ROM functional, the outputs and chip are enabled by their respective pins. Since the output pins of this chip can also act as inputs, write enable is inhibited. The output of this is displayed by LEDs, but is also wired to another ROM chip: the control ROM.

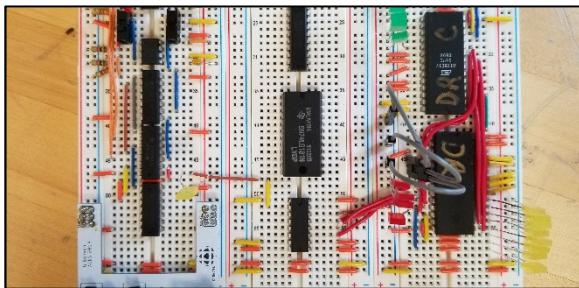
This chip is not programmed correctly (currently), but is wired nearly identically to the program ROM. The only difference is that the eight output pins of the first ROM connect to the first eight address pins of the control ROM. In the future, this chip will be connected to the rest of the of the processor and will distribute data, but for now it will remain a blank slate. LEDs are inserted on the output pins to display the outgoing data, and the first section of the circuit is finished.

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
000000:	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000010:	FF	FE	FD	FC	FB	FA	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0
000020:	FF	00														
000030:	FF															
000040:	AA	55	.U.U.U.U.U.U.U.U														
000050:	FF															
000060:	FF															
000070:	FF															
000080:	FF															
000090:	FF															
0000A0:	FF															
0000B0:	FF															
0000C0:	FF															
0000D0:	FF															
0000E0:	FF															
0000F0:	FF															

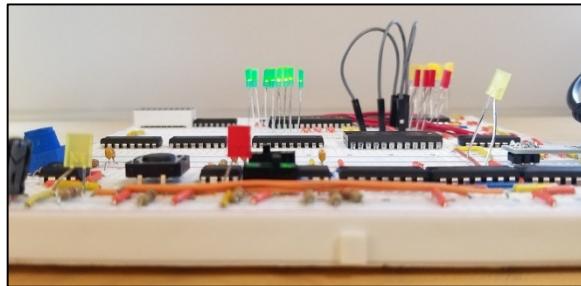
Program EEPROM Registers

Media

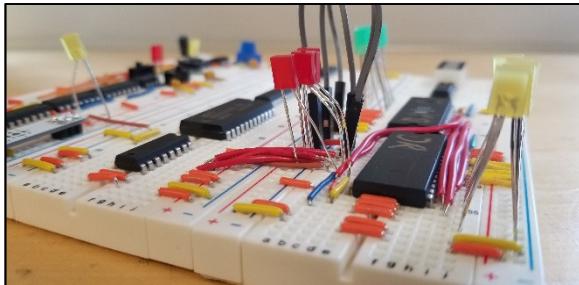
Personal Video: <https://youtu.be/rU2Jw-Tx6j4>



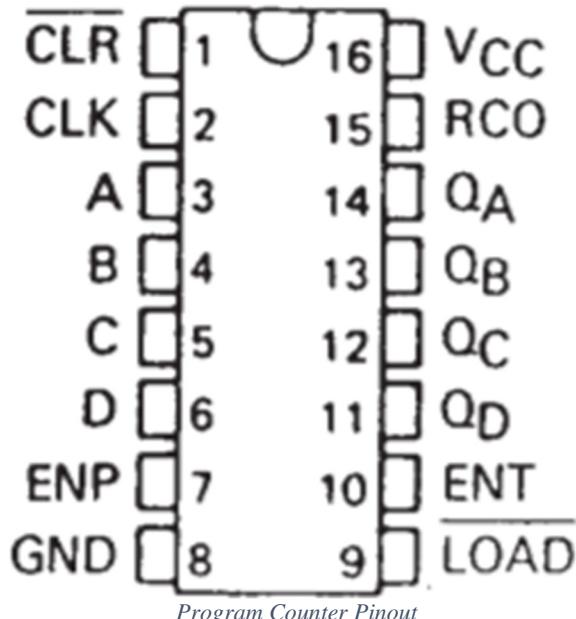
Full Processor



Bimetric View



Trimetric View - EEPROM



Reflection

After this project, this computer seems a whole lot less daunting. The first time we were told that in two weeks we would be finished the entire CHUMP, I panicked a little bit. I understood the general logic behind the computer, but when it came down to the actual wiring, I was at a loss. Now, I understand the reasoning behind each wire. This is most likely the easy part of the CHUMP, but now I feel a lot more under control than I did a week ago.

Part E. The Completed Processor

Purpose

This is the final circuit of the CHUMP. The purpose of it is to bring everything together and create a completely working four-bit computer made entirely of chips and wires.

Reference

Home Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/4BitComputer/index.html#Individual>
 RAM: <https://www.silicon-ark.co.uk/datasheets/sn74ls189an-datasheet-texas-instruments.pdf>
 Accumulator: <http://www.ti.com/lit/ds/symlink/sn74ls377.pdf>
 Address: <http://www.ece.sunysb.edu/~dima/74ls174.pdf>
 Multiplexer: <http://www.ti.com/lit/ds/symlink/sn74ls157.pdf>

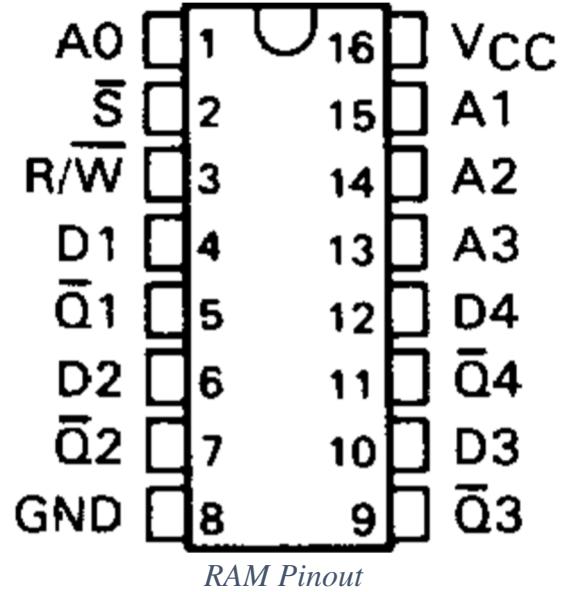
Theory

The control ROM receives the opcode from the program ROM. This opcode is the instruction for the entire computer. Using this information and its eight output pins, the control ROM distributes the proper ALU instruction, which function the RAM should be performing (read or write), and the inhibiting pin of the accumulator chip (Accum). It knows this based on the previously agreed upon interpretation/language inside the registers of the ROMs. For example, the current opcode for an addition operation is 0010. This number becomes the address for the control ROM. It looks inside that specific register and outputs whatever it sees. In this case, it would set the ALU to addition and enable the Accum. Since we aren't using the RAM, it is set to read as a precaution.

The RAM that is used in this project is the SN74LS189N. It has 16 four-bit registers. As a result, it has four address pins, four data input pins, and four output pins. It works similarly to the EEPROMs as previously discussed, the main difference being that in this project we are writing to it in certain scenarios. To write to this chip, the R/W pin is grounded.

The address chip is a hex flip-flop. It has eight inputs and eight corresponding outputs. When an input is given power, even for an instant, its respective output is set to high. In this way, it acts as a single register that holds the address for the RAM. Once its clock pin completes one full cycle, all its outputs are set low. The Accum is the exact same chip, but it is an octal flip-flop with an inhibitor.

The multiplexer has two primary inputs comprised of four pins each: A and B. Based on the input of a select pin, it decides to output either all the A inputs or all the B inputs. This is used in the CHUMP to determine whether to use the constant provided by the program ROM or the output of the RAM.



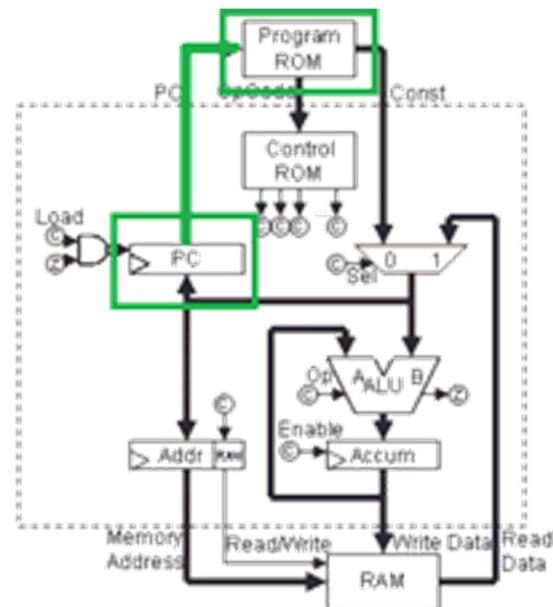
Procedure

To begin, the control ROM is flashed to contain the correct opcode translations (as this was not the case previously). For example, the addition instruction needs the ALU set to add and the Accum enabled. This means the select pins are, in order, high, low, low, and high. Mode is also set to low and C of n is set to high. This creates the control code 1001 0101. As shown by the [control table](#), R/W is set to read, not because reading is necessary to this computation, but because the RAM should not be written to at this time. This table shows the outputs of all the control ROM address, so this is the information that is flashed to each address, with the opcode indicating the address for each register.

This all works, but there is one problem: other chips need to be told what to do. The multiplexer needs to select which input to display on its output, so it is wired on the LSB of the opcode; if it is a zero it chooses the constant and if it is 1 it chooses the RAM. This only works because of the way the opcodes are arranged. The PC also needs a control pin to tell it when to load in a new value. For this, logic gates are used. Specifically, the middle bits of the opcode are OR'd together and AND'd with the MSB of the opcode. This tells the computer whether it is performing a GOTO or an IFZERO command. This is then NAND'd with the 'A=B' pin of the ALU. This pin, despite its name, does not compare the inputs. "The A = B output from the device goes HIGH when all four F outputs are HIGH" (ALU Datasheet). Knowing this, the ALU is told to output all ones when performing a GOTO, and 'Not A' when performing an IFZERO. If 'A' is zero, it will set the 'A = B' pin high, loading the PC like originally intended.

Once all the EEPROMs are flashed correctly, the chips are laid out onto three breadboards. By studying the block diagram, it is possible to determine which chip connect, so the chips with the most connections are as close to the center as possible. The lower nibble of the program ROM output is wired to the A input of the multiplexer, as this has the possibility of being used in the ALU's computation. The output of the multiplexer is wired to three chips: the PC, the address register, and the B-input of the ALU. The ALU uses these four bits for the computation. The address register changes to match this output in case the RAM is read from and this is to be used as an address for a register in RAM. Finally, the PC is wired to this output in case there is a GOTO or an IFZERO and a new value needs to be loaded to it. The output of the RAM is wired to the B input of the multiplexer. The control ROM is wired in order of the control table. The RAM control, however, is wired through the hex flip-flop to stick with the clock cycle.

Parts	Quantity
AT28C17 (EEPROM)	2
SN74LS189N (RAM)	1
SN74LS181N (ALU)	1
(PC)	1
SN74LS377 (Accum)	1
(Multiplexer)	1
(Flip-Flop)	1
2.2KΩ resistors	5

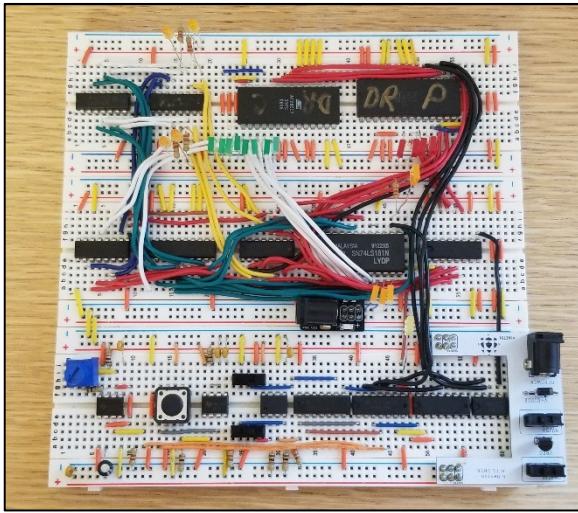


Media

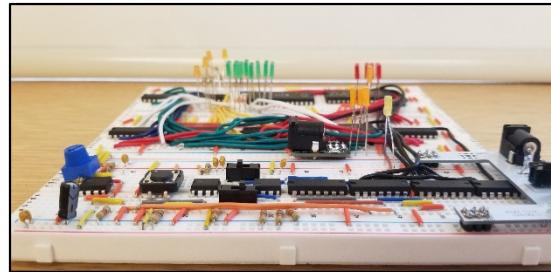
Personal Video: <https://youtu.be/dj4JpOweLYQ>

Instruction	OpCodes	ALU	S ₃	S ₂	S ₁	S ₀	M	Cn	Acc	R/W
LOAD const	0000	B	1	0	1	0	1	X	0	X
LOAD IT	0001	B	1	0	1	0	1	X	0	1
ADD const	0010	A PLUS B	1	0	0	1	0	1	0	X
ADD IT	0011	A PLUS B	1	0	0	1	0	1	0	1
SUBTRACT	0100	A MINUS B	0	1	1	0	0	0	0	X
SUBTRACT IT	0101	A MINUS B	0	1	1	0	0	0	0	1
STORETO	0110	ignore	X	X	X	X	X	X	1	0
STORETO IT	0111	ignore	X	X	X	X	X	X	1	0
READ const	1000	ignore	X	X	X	X	X	X	1	1
READ IT	1001	ignore	X	X	X	X	X	X	1	1
GOTO const	1010	Logic 1	1	1	0	0	1	X	1	X
GOTO IT	1011	Logic 1	1	1	0	0	1	X	1	X
IFZERO const	1100	Not A	0	0	0	0	1	X	1	X
IFZERO IT	1101	Not A	0	0	0	0	1	X	1	X

Control Code Table



Complete CHUMP



CHUMP - Bimetric

Reflection

I loved this project; The wiring was intense and stressful, but also fun and meditative. The block diagram for the CHUMP contains the minimal amount of information required to completely wire this circuit up; there was no paint-by-numbers, but it still showed the flow of information perfectly. It has been a thrill ride of a project and guaranteed my favourite project we have ever done in hardware. The moment it worked, I was up in the clouds. Thanks to the chump club, I have a memory I will treasure for ever. The boys and the coursework here are truly nothing short of spectacular. I can't wait to see what's in store for us next.

Project 18. Short ISP: Video Turntable

Purpose

This photographic turntable is made to allow for 360 videos of projects for the entire Design Engineering Studio (DES). It also serves a secondary purpose as a rotary display for students to view projects on, complete with a character LCD to show the name of the current project atop the turntable.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/1819/ISPs.html#logs>

Colour Sensor Datasheet: <https://www.mouser.com/catalog/specsheets/TCS3200-E11.pdf>

Barcode Reference: https://www.barcoderesource.com/code128_barcodefont.html

Theory

To determine what is displayed on the LCD, the colour sensor acts as a barcode scanner. The motor spins the platform, allowing the barcode to pass by the sensor. In this process there are multiple different techniques being used.

Barcodes themselves have multiple different ways of storing information for their various uses, all with many pros and cons. In this project, the encoding font of choice is barcode 128B. This font is perfect for many reasons. Firstly, 128 is an international standard, so finding barcode generators online is incredibly easy. Secondly, it has the entire range of alphanumeric characters, as well as 32 other punctuation symbols. It is essentially the entire ASCII table without all the low-level commands. This makes it easy and efficient while still maintaining all the required functionality. Finally, it lines up perfectly with the ASCII table, allowing for easy conversion.

To scan the barcode, there are a few different options. IR sensors, colour sensors, and barcode scanners themselves can all tell the difference between black and white. While a barcode scanner is most likely the best solution for gathering this specific information, they are expensive. By taking advantage of the rotating platforming we can use any device that can read the difference between black and white. For this, a TCS3200 colour sensor is used. It can read the three primary colours: red, green, and blue (as well as clear). It has a total of eight pins: power and ground, an enable, four select pins, and one output pin. Two of the select pins determine which colour it is analyzing. The other two tell the sensor how sensitive it should be. The output pin, however, is a little strange. The output of the TCS3200 is not a linear colour reading, but rather a wavelength that encodes the intensity of a specific colour. Because of this, a simple `analogRead()` will not be used, but rather another function called `pulseIn()`. This tracks the length in times in milliseconds that a voltage is either high or low (depending on one of the parameters).



Colour Sensor Photodiodes

The TCS3200 colour sensor has an eight-by-eight array of light sensitive photodiodes, each with a specific colour filter. Each colour it can detect uses 16 of the photodiodes (red, green, blue, and clear).

The motor chosen for this project is rather tall and would be better if it were parallel to the ground. To fix this problem, bevel gears are used. There are just like normal gears with the minor difference that they are used at a 90-degree angle to one another. These gears also allow for slight tweaking in the balance of speed vs torque, which is useful tool to have.

The motor used is a 0.79 Nm brushed DC motor with a 4mm D-shaft. Newton-meter is a measurement of torque; in other words, this component can lift nearly eight kilograms one centimeter away from its axis of rotation. This is evidently enough power to spin the platform with nearly anything that can fit on it. The D-shaft allows for easy design, as a bevel gear can be designed to friction fit onto it without any extra components.

Procedure

To begin, the 3D design for the prototype is made. It has a platform connected to a bevel gear system that is driven by the motor. The motor is screwed into the base which holds these two parts together with a disk joint. The base is also designed to hold the LCD as well as the colour sensor, which needs to be close to the edge of the platform.

Once the design is printed, the motor is screwed in and connected to a tristate toggle switch. This acts as a physical half H-bridge, allowing current to flow through the motor in either direction, depending on the state of the switch. The power of the switch is connected to a potentiometer which is then connected to 12 volts. This is the maximum recommended voltage for the motor.

To keep this circuit on one power supply, a 5V voltage regulator is used to power the ATtiny84. This is the microcontroller that is used as it has the perfect number of pins for this project. There are two main devices attached to the MCU: a character LCD and a colour sensor. These are the devices that read and display the barcode information. Since a normal LCD requires way too many pins that the ATtiny84 simply does not have, an LCD with I2C is used. Thus, the LCD is connected to power, ground, SDA, and SCL. The colour sensor is hard-wired to have a sensitivity of 20%. The other two selects and the output pin are connected to the MCU. This allows the user to take control over which colour is being read.

Unfortunately, there is no datasheet for the I2C LCD, so a library must be used. However, all the libraries are for the old form of I2C, so some of the libraries are edited. The colour sensor is made to calculate the rotary speed of the barcode (and the platform), and then scan a barcode that has 0.8326mm per bar. It then concatenates all the information into a string that is then displayed on the LCD.

Parts	Quantity
ATtiny84	1
0.79 Nm Motor	1
I2C 16x2 Character LCD	1
TCS3200 Colour Sensor	1
ON-OFF-ON Toggle Switch	1
10KΩ Potentiometer	1
Voltage regulator	1
Disk joint	1

Code

```
// Author: Daniel Raymond
// Date: September 14, 2018
// Description: Code for the display of the turntable. It reads a barcode
// and displays the result on an I2C LCD.

#include <TinyWireM.h>
#include <LiquidCrystal_I2C.h>

#define SDA A6 // 7:PA6
#define SCL A4 // 9:PA4

#define S2 A1 // 12: PA0
#define S3 A0 // 13: PA1
#define clrPin A2 // 11:PA2
#define scanBtn A3 // 10: PA3

#define adrs 0x3F
#define radius 60
#define pi 3.14159
#define codeSize 187 // barcode length in mm
#define bitWidth 0.8326

LiquidCrystal_I2C lcd(adrs, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);

uint8_t asciiChar = 0;
String text = "";
uint8_t lngth = 0;
uint8_t clr = 0;

void setup() {
    pinMode(S2, OUTPUT);
    pinMode(S3, OUTPUT);
    lcd.begin(16, 2); // Dimensions of the LCD
}

void loop() {
    if (scanBtn) {
        scanCode();
        printName();
    }
}

// Calculates the outer speed of the disk in mm/s
unsigned long rotationSpd() {
    unsigned long startTm = millis();
    unsigned long endTm;
    unsigned long rotationSpd;

    while (!readFreq(LOW, HIGH, 0, 1)); // Wait for red line to appear
    delay(50); // Wait for red line to pass

    while (!readFreq(LOW, HIGH, 0, 1)); // Wait for red line to appear again
    endTm = (millis() - startTm) << 1; // Timing a full rotation
```

```
rotationSpd = 2000 * pi * radius / endTm; // platform speed in mm/s
return rotationSpd; // gives circumference speed in mm/s

}

// Reads in the barcode
void scanCode() {
    text = "";
    unsigned long rSpd = rotationSpd();
    unsigned long startTm = millis();
    unsigned long endTm = millis() + (codeSize / rSpd);
    float bitTime = bitWidth / rSpd; // time per each bit

    // Wait for barcode
    while (!readFreq(LOW, HIGH, 0, 1)); // Waits for red to spike

    // Scan barcode
    while (millis() != endTm) {
        for (uint8_t x = 0; x < 8; x++) { // Read in 8 bits for a character
            asciiChar += readFreq(HIGH, HIGH, 0, 1) << (7 - x); // create a byte
            delay(bitTime);
        }

        text += (char)(asciiChar + 32); // Using barcode font 128B
        asciiChar = 0;
    }
}

void printName() {
    lcd.clear(); // Resets lcd
    length = text.length();
    lcd.setCursor((16 - length) >> 1, 0); // Centers text
    lcd.print(text); // Prints name
}

// Reads in an intensity from the colour sensor
int readFreq(uint8_t p2, uint8_t p3, uint8_t minimum, uint8_t maximum) {
    int tempFreq = 0;
    // Setting RED (R) filtered photodiodes to be read
    digitalWrite(S2, p2);
    digitalWrite(S3, p3);

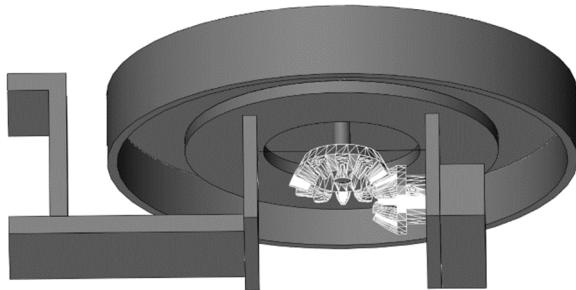
    // Reading the output frequency
    tempFreq = pulseIn(clrPin, LOW);

    // Remapping the value of the frequency to white or black
    tempFreq = constrain(tempFreq, 17, 260); // Measured max and min
    tempFreq = map(tempFreq, 17, 260, maximum, minimum);

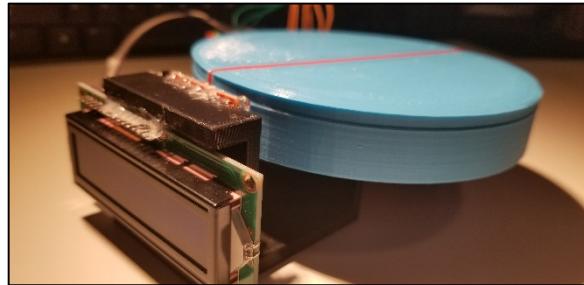
    return tempFreq;
}
```

Media

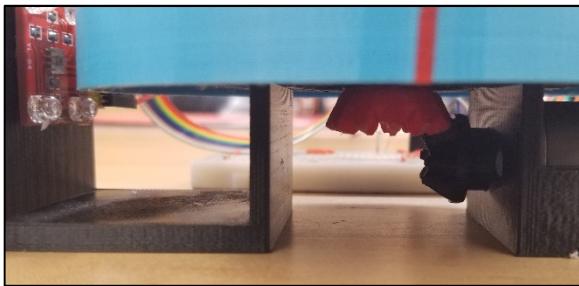
Personal Video: <https://youtu.be/aVvzP7ODx0s>



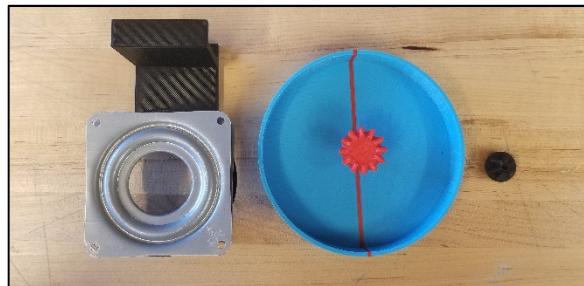
Complete CAD Design



I2C LCD Display



Sensor & Gears



3D Printed Parts

Reflection

Although there were times where it seemed impossible to complete, I am proud of what I've managed to accomplish. The core problem with it seems to be that the gears do not mesh well enough to provide a consistent enough rotational speed. This affects the primary usage, as well as the colour sensor. Despite this, I did learn a surprising amount. The colour sensor is a very consistent and useful part that I can see myself using in the future.

Project 19. Keypad Storage Unit

Purpose

This project is a storage unit that limits access to one compartment at a time. It is a personal vending machine, of sorts, that places a primary role on the use of the telephone keypad.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/1819/Tasks.html#TelephoneKeypad>

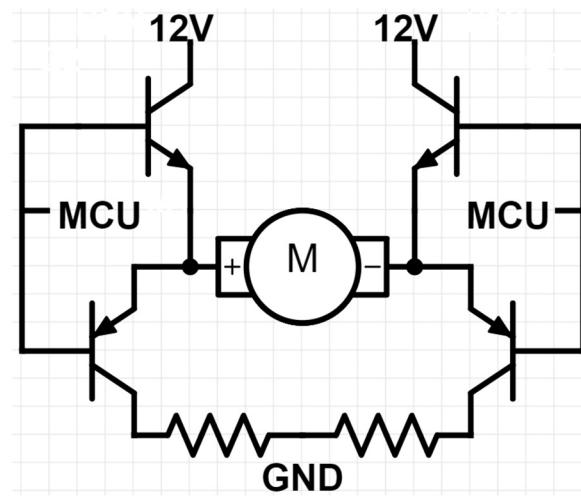
H-Bridge Datasheet: <http://www.ti.com/lit/ds/symlink/sn754410.pdf>

Theory

A software solution is used to debounce the keypad. The keypad itself does not have a power supply, but simply connects corresponding row and column pins together. To measure a button press, the rows are continually outputting low. When a button is pressed, the corresponding column pin, along with the common, is pulled low. Once this is detected, a timer waits for the bouncing to end. At this point in time the button is still pressed so it reads all the column pins, which reside in port B. To determine which column was pressed, a switch case statement is used. When the column is found to be grounded, it is the correct one. Next, the row pin must be determined. To do this, a loop runs which grounds each row pin individually. When the correct row pin is grounded, it triggers an if statement which returns the correct button value. Finally, when the button is released, another delay runs, ignoring the debouncing period. Since the bouncing interval when being released is longer than the press, the interval is made twice as long.

To control the motor, an H-bridge is used. This controls the direction the current travels, causing the motor to spin in either direction. When a single side of the circuit is powered, the NPN on its side is turned on while the PNP is turned off. This forces the current to flow through the motor. Doing this for the other side causes the current to flow through the motor in the opposite direction. When both pins are turned off, the motor is stationary.

The motor that is used to shift the canvas is a brushed DC motor. This component works in a very interesting way. When the motor is powered, there is a brush that contacts a metal plate that is connected to an electromagnet. This causes the magnet to become magnetic and be attracted to a ferromagnet that lines the exterior of the motor. This causes the shaft to rotate. This act disconnects the brush from the metal plate and connects it with a plate on the opposite side. This reverses the poles of the two internal electromagnets, causing it to be attracted to the opposite side of the motor. This cycle continues until power is disconnected.



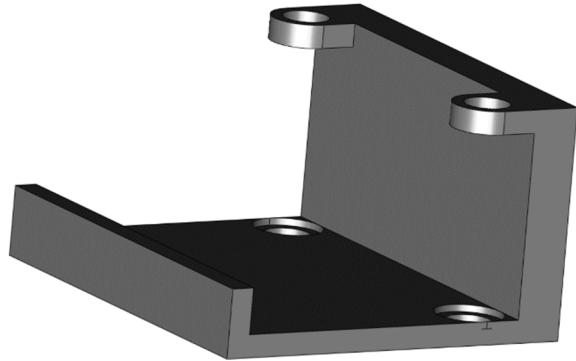
H-Bridge Schematic

Procedure

This project is heavily design centric. There are six main components to the tower: levels, inserts, upper supports, the bottom support, the roller, and the inverter stand. The levels comprise the main build of the design, being the landings for each stored item. This component, along with the inserts, create a modular design to increase the height of the build. The only special floor is the top, which has a curved roof.

This helps with the movement of the belt. The upper supports are to protect the stored items inside as well as provide structural support. The bottom support, however, is special. It provides the connection between the tower and the motor. There are holes for M3 screws to connect the motor, and an insert to hold the roller as it rotates. Unfortunately, the is too much resistance to pull the canvas using the roller; no amount of tension helps this problem. The resulting solution is the inverter stand. The idea is to allow gravity to help with the problem, not add to it. By flipping the entire tower, the friction primarily resides on the roller. The motor is connected to an external power source. The Arduino controls an H-bridge IC that allows for bidirectional control of the motor. The debounced keypad is read and compared with the current level of the opening. The motor then shifts the canvas cover accordingly, opening the correct storage compartment.

Parts	Quantity
Arduino UNO	1
0.36 Nm Motor	1
H-Bridge	1
DC Jack	1
Canvas Cover	1
3D Printed Components	18
M3 Screws	2



Level Module

Code

```
// Author: Daniel Raymond (Keypad Code provided by Chris D'Arcy)
// Project: Sectional Storage Unit
// Date: 2019-11-31
uint8_t keys[4][3] = {{1, 2, 3}, // There are only 6 floors
                      {4, 5, 'x'},
                      {'x', 'x', 'x'},
                      {'x', 0, 'x'}}
;
uint8_t colPins[] = {9, 10, 11};
uint8_t numCols = sizeof(colPins);
uint8_t rowPins[] = {4, 5, 6, 7};
uint8_t numRows = sizeof(rowPins);
uint8_t row, col; // Looped variables
#define INTERVAL 10L // Debouncing wait period
#define COMMON 2
unsigned long detect; // Keeps track of time during debounce
uint8_t input; // Stores column values when read
uint8_t level = 0; // Current floor the opening is on
uint8_t destination; // Floor that needs to be open
#define FRWPIN 12 // Motor pins
#define BACKPIN 13
#define TMCONSTANT 1500 // Scaling the motor's duration
```

```

void setup() {
    pinMode(COMMON, INPUT_PULLUP);           // Connected to every column
    for (col = 0; col < numCols; col++)
        pinMode(colPins[col], INPUT_PULLUP); // Every column starts high

    for (row = 0; row < numRows; row++) {
        pinMode(rowPins[row], OUTPUT);       // For 'scanning' each row with grounds
        digitalWrite(rowPins[row], LOW);
    }
    pinMode(FRWDPIN, OUTPUT);
    pinMode(BACKPIN, OUTPUT);
}

void loop() {
    while (bitRead(PIND, COMMON));           // Waits until common pin is turned low
    detect = millis();                      // Reads time
    while ((millis() - detect) < INTERVAL); // Waits until interval has passed
    input = PINB;                          // Reads columns pins
    destination = getTelKey(input);         // Record key input

    // Do nothing if * or # is pressed, or if it is already at the correct floor
    if (!(destination == '*' || destination == '#') && (level != destination))
        moveMtr();                         // Move the motor the corresponding amount

    while (!bitRead(PIND, COMMON));           // Waits until any button is released
    detect = millis();                      // Reads time
    while ((millis() - detect) < INTERVAL << 1); // Waits for bouncing interval to pass
}
char getTelKey(uint8_t input) {
    switch (input & 0b00000110) {           // Masks every pin but the columns
        case 0b00000110: col = 2; break;   // Checks which column is being grounded
        case 0b00000101: col = 1; break;
        default: col = 0;                  // If it isn't the other 2, it must be this
    }
    for (row = 0; row < numRows; row++) {
        digitalWrite(rowPins[row], HIGH);   // Ignores every row...
        if (digitalRead(colPins[col])) {
            digitalWrite(rowPins[row], LOW);
            return keys[row][col];
        }
        digitalWrite(rowPins[row], LOW); // ...Except for this one (ground each row pin)
    }
}

void moveMtr() {
    int8_t dist = destination - level; // Determine # of floors to travel (signed)

    if (dist > 0)                     // Determine direction for the motor to spin
        digitalWrite(FRWDPIN, HIGH);
    else
        digitalWrite(BACKPIN, HIGH);

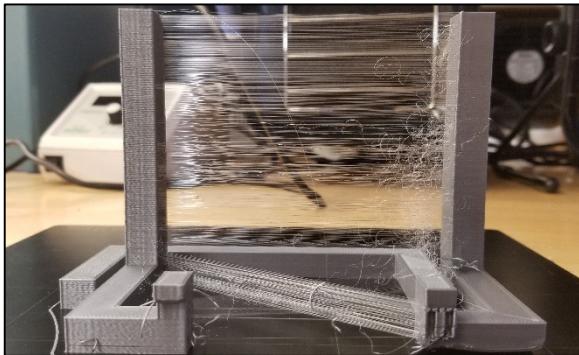
    delay(abs(dist)*TMCONSTANT);      // Wait for the motor to move the right distance

    digitalWrite(FRWDPIN, LOW);
    digitalWrite(BACKPIN, LOW);
    level = destination;             // destination has now arrived
}

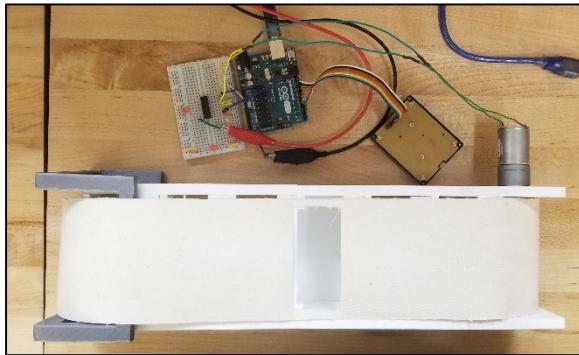
```

Media

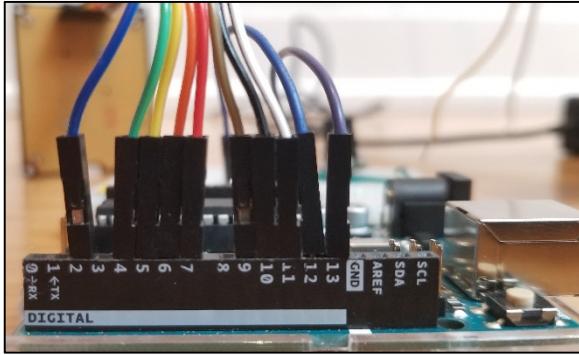
Personal Video: <https://youtu.be/QzXRI49Na74>



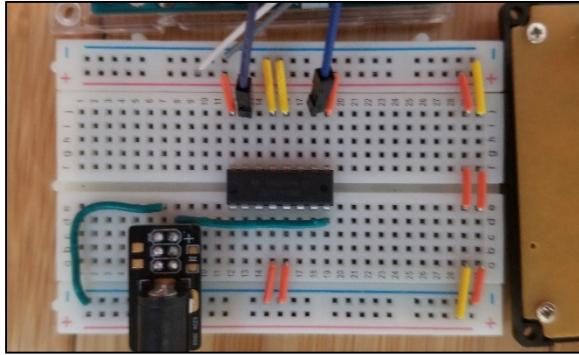
Inverter Stand



Complete Design



Arduino Pinout



H-Bridge Circuit

Reflection

This project has increased my design capabilities immensely. I have always had a difficult time knowing where to start, and this storage unit has shown me that it is better to start to build the general concept first. Having multiple iterations of a design is important; the finer details of a project can always be changed but starting and noticing problems early is crucial. The keypad is an interesting component and it confused me for a good while. I found myself writing my own keypad code simply to better understand the device, even if it wasn't used in the final design. In the end, I am disappointed that the canvas roller was not as predictable as I'd like, but some additional modifications might fix the issues.

Project 20. Rotary BCD Switch

Purpose

This is the first major assembly project. In it, a hexadecimal rotary BCD switch (rBCD) determines the output of a seven-segment display, entirely programmed in AVR Assembly. For example, if the rBCD is turned to six, a six will appear on the display.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/1819/Tasks.html#Rotary>

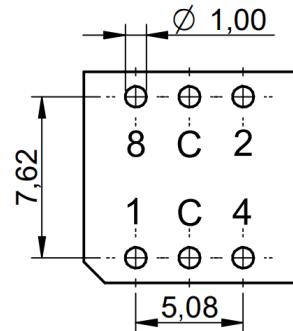
AVR Assembly Instructions:

https://www.microchip.com/webdoc/avr assembler/avr assembler.wb_instruction_list.html

Theory

Rotary BCD

The new component involved is the rBCD. As the name implies, it encodes a number in four of its pins to make a nibble of data. To do this the rBCD has six pins: four-bit pins and two power pins (presumably for symmetry's sake). When a hexadecimal value is pointed to, its binary equivalent is connected to power. This alone cannot be read from, as there is no current flow. Instead, there is a resistor tying these pins to ground. This current is then read in by the Arduino.



Rotary BCD Pinout: LSB-MSB

Arrays

To light up the correct segments, an array is used. In each of its cells, it contains a byte of information that, when put to the output register of the correct port, will light up the seven-segment display. The information corresponds to its address in the array. To program this in assembly, there are a few requirements. First, it needs to know where you are storing the information, whether it is in program flash or in RAM. While program flash is plentiful, it is slower for the microcontroller to access (three cycles versus one). Next, it needs to have a name assigned to the array. This simply allows the programmer to reference the first address of the array, as it would otherwise be unknown. To access this information, a pointer is used. Pointers are 16-bit words comprised of two accumulators. When given to the 'lpm' (load from program memory) or the 'lds' command (load from SRAM), they are treated as an address that a value should be loaded from.

Procedure

The circuit itself is relatively simple. To begin, the rBCD is connected to power. Its bit pins have pulled down resistors and are wired to the MCU in order of significance. It is important that they are all tied to the same port of the

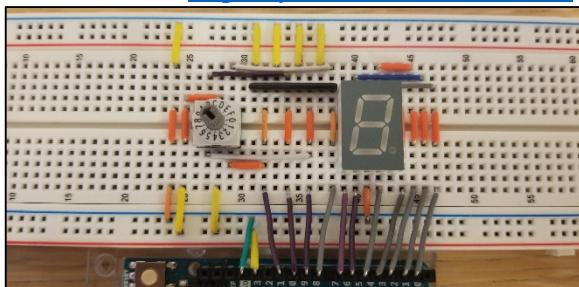
Arduino, in this case it is port B. Since the seven-segment display is a common cathode component, pins three and eight are grounded. The rest of the display's pins are connected to port D of the microcontroller.

Parts	Quantity
Arduino UNO	1
Rotary BCD Switch	1
10K Ω Resistor Network	1

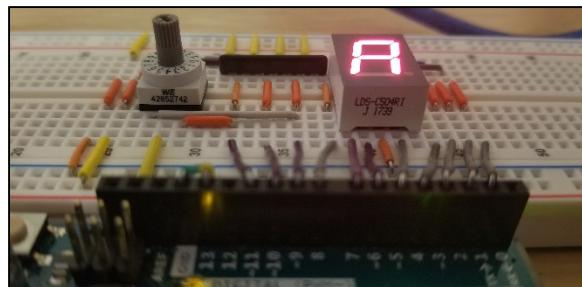
To develop a general idea for the coding of the device, a working program is made using mid-to-high-level techniques. In this iteration, port manipulation is used to allow for an easier conversion to assembly. The program holds an array, and when the rBCD is read, the value given is used as the address for the array. The value in each cell corresponds to the segments that should light up on the display for that value. For example, if the value read in by the rBCD was *1000*, then the data in cell eight is used. In this case, eight should light up all the segments, so it would contain *0xFF*, or 255. Each bit in this byte triggers a specific pin of the display. After programming the project in C, converting AVR Assembly is quite easy, as it uses the exact same concepts and framework.

Media

Personal Video: <https://youtu.be/JNii5Ev57Ko>



Completed Circuit



Working Circuit with Pin Numbers

Reflection

This project has demonstrated to me the power of assembly language. This is something that I have been looking forward to learning for a long time. I always had a feeling that it would be something that I would enjoy, but now it is finally here. While assembly might not have an immediate application, I love knowing how things work ‘under the hood.’ It is fantastic this opportunity is given to me in the first place.

Code

```
#include <avr/io.h>

a = (1 << PD3)           ; Segment pins
b = (1 << PD4)
c = (1 << PD5)           ; Each control a respective bit
d = (1 << PD6)           ; in a byte
e = (1 << PD7)
f = (1 << PD2)
g = (1 << PD1)

pwr    = PB5               ; Power for the rotary BCD
sPins  = a|b|c|d|e|f|g   ; Direction Register for the 7-Segment
sDDR   = DDRD-0X20          ; Direction for the 7-Segment
sOUT   = PORTD-0x20         ; Out port for the 7-Segment
rDDR   = DDRB-0x20          ; Direction for the Rotary BCD (rBCD)
rIN    = PINB-0x20          ; In port for the rBCD
rOUT   = PORTB-0x20         ; Out port for the Power pin

digits: .byte  a | b | c | d | e | f      ; 0
        .byte  b | c                   ; 1
        .byte  a | b |     d | e |     g ; 2
        .byte  a | b | c | d |           g ; 3
        .byte  b | c |                 f | g ; 4
        .byte  a |     c | d | f |     g ; 5
        .byte  a |     c | d | e | f | g ; 6
        .byte  a | b | c             ; 7
        .byte  a | b | c | d | e | f | g ; 8
        .byte  a | b | c |           f | g ; 9
        .byte  a | b | c |           e | f | g ; A
        .byte  c | d | e | f |     g ; B
        .byte  a |     d | e | f       ; C
        .byte  b | c | d | e |     g ; D
        .byte  a |           d | e | f | g ; E
        .byte  a |           e | f | g ; F

.global setup
setup:
    ldi    r16, sPins      ; Prep every pin in port D to output
    out    sDDR, r16        ; Set Direction Register to all output

    sbi    rDDR, pwr        ; Set every pin to input except pin 13
    sbi    rOUT, pwr        ; Set pin 13 high (but nothing else)
ret

.global loop
loop:
    in    r29, rIN          ; Read the entire port that the rBCD is on - put it on ZL
    andi r29, 0x0F          ; Only read the lower nibble (4 pins of rBCD)

    ldi    ZL, lo8(digits); Set initial pointer to base value of the array
    ldi    ZH, hi8(digits); Set upper byte just in case the array is stored in a
high location
    add    ZL, r29           ; Increment pointer by the address

    lpm   r16, Z            ; Load the value of SRAM at Z into r17
    out   sOUT, r16          ; Output value on the 7-segment
ret
```


Project 21. Knight Rider

Purpose

The goal of this project is to program a shift register in assembly. Its output leads to an LED bar graph. Specifically, the bar graph is supposed to follow an incrementing pattern, adding the next LED to all the previous ones. The second animation is a binary decrementing pattern. A switch decides which animation to show.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/1819/Tasks.html#KnightRider>

Tim Morland's PCB: <http://darcy.rsgc.on.ca/ACES/PCBs/index.html#ShiftBar>

Personal Website: dan.raymond.ch

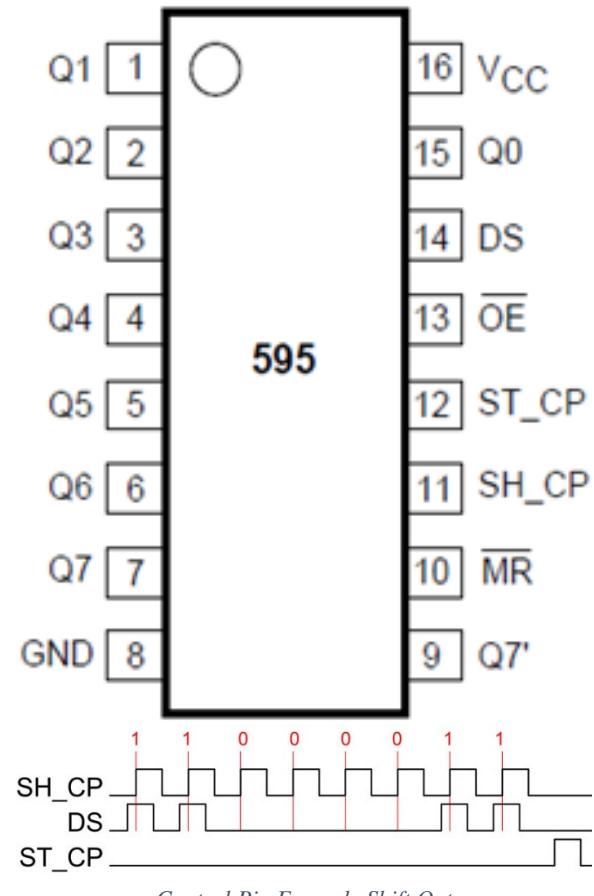
Theory

The shift register is a serial-to-parallel IC with three main control pins: latch, data, and clock.

The latch controls whether the chip is reading or presenting the data. To begin reading, this pin is set low. After this, a value of either high or low is presented on the data pin. When the clock pin goes high (on a RISING edge) the shift register reads the data pin and ‘loads’ that value into the byte. After this process cycles eight times (or more when daisy chaining) the latch is pulled high, displaying the byte on the shift register’s eight output pins.

The data byte presented by the shift register is made to follow the pattern $2^n - 1$, so all the LEDs stay on as a new one is added onto it. To present this data, as mentioned before, the data pin needs to present each bit individually. To accomplish this, a 1-bit mask cycles through and is AND’d with the data. If the result is zero, the data pin is grounded; otherwise it is pulled high. To create this data byte there are two registers involved. The first starts at one and then, through every iteration of the loop, shifts left one bit. At each cycle this value is AND’d with the data byte that is sent out to the shift register.

The final piece is the switch. It is connected to PCO-2 of the Arduino, with its outer pins tied to power and ground respectively. The middle pin simply reads the value that it is connected to and sets the animation.



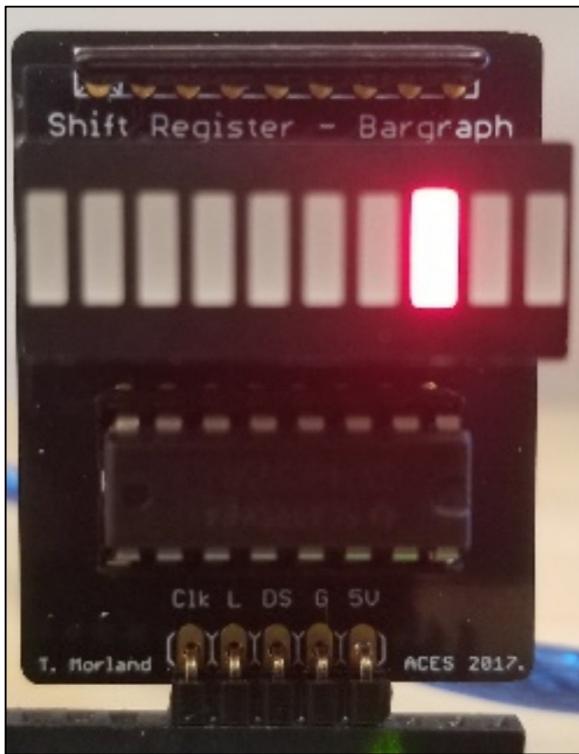
Control Pin Example Shift Out

Procedure

The creation of this device is incredibly simple. It uses Tim Morland's Arduino Appliance to attach the shift register to the LED bar graph. Everything is soldered (either through a chip seat or directly), and then the PCB is inserted into the Arduino. Power and ground are applied to their respective pins, and the appliance is programmed to shift out data that is displayed on the LEDs, as described in the [Theory](#) section.

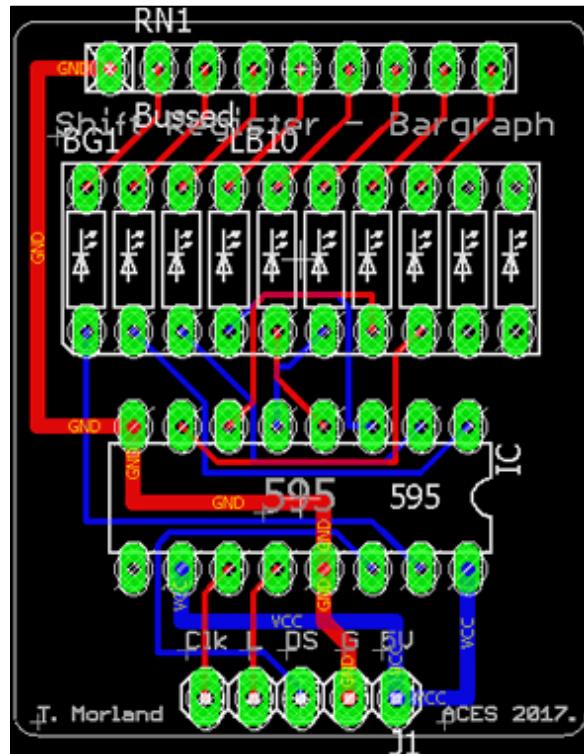
Media

Personal Video: <https://youtu.be/5mUv4wJ75pQ>



Working Shift Register Bar Graph

Parts	Quantity
Arduino UNO	1
T. Morland PCB	1
LED Bar Graph	1
220 Ω Resistor Network	1
74HC595	1
Right Angle Header Pin	5



PCB Board Design

Reflection

I love assembly. There are very few aspects of it that inhibit use; you are free to build anything because literally every part of the MCU is at your disposal. Having access to each register and being able to change individual bits is a skill that I did not realize I wanted, but now am glad I have. This project has demystified lots of the high-level functions that I once took for granted (shiftOut, SPI, etc); I can't wait for the next.

Code

```

; PROJECT: Knight Rider (shift register animations)
; PURPOSE: To manipulate a shift register (SR) in AVR Assembly
; DEVICE: Arduino (ATmega328p)
; AUTHOR: Daniel Raymond
; DATE: 2019-02-01

.org      0x0000      ; Tells PC that reset is the 1st instruction
rjmp    reset

.equ    DDR          = DDRD
.equ    POUT         = PORTD

.equ    clock        = PD7
.equ    latch        = PD6
.equ    data         = PD5
.equ    gnd          = PD4
.equ    pwr          = PD3
.equ    outPins      = (1<<clock | 1<<latch | 1<<data | 1<<gnd | 1<<pwr)

.equ    swtchPwr     = PC0
.equ    swtchRead    = PC1
.equ    swtchGnd     = PC2
.equ    DDRC          = DDRC
.equ    POUTS         = PORTC
.equ    PIN           = PINC
.equ    outPinSwtch  = (1<<swtchPwr | 1<<swtchGnd)
.equ    onPos         = (1<<swtchPwr | 1<<swtchRead)

.def    util    = r16      ; r16 is a general-purpose register
.def    read    = r17      ; Reading in the value of the switch port
.def    incr    = r18      ; Incrementing register for the # of LEDs that are on
.def    leds    = r19      ; The register that holds the final output of the SR
.def    mask    = r20      ; Mask to isolate which LEDs need to be on
.def    bit     = r21      ; The bit in focus (when shifting out)

reset:
    ldi    util,    outPins      ; Set pins on the SR to output
    out    DDR,     util        ; Change data direction to match outputPins
    ldi    util,    outPinSwtch ; Sets up power and ground for the switch
    out    DDRC,   util        ; Makes data direction match
    sbi    POUT,   pwr         ; Set the power pin to high
    sbi    POUTS,  swtchPwr    ; Give the switch power

    ldi    incr,   1            ; Set the first LED on

loop:
anim1:
    in    read,   PIN          ; Read the switch
    cpi   read,   onPos        ; If the read pin is high
    brne  anim2,  anim2        ; If the switch is low, go to the 2nd Animation
    ldi   incr,   1            ; Revert the increment register
    clr   leds,   0             ; Turn off all LEDs (LSB will be turned on)
    rcall  rowLEDs, rowLEDs   ; Otherwise, it will run the row anim
    rjmp   anim1,  anim1        ; Once it is done, it will run the row anim again

anim2:
    ldi   leds,   0xFF        ; Start the leds as all on
    rcall  decr,   decr        ; Run the animation
    rjmp   loop,   loop        ; Go back to the start of the loop
    rjmp   reset,  reset

```

```

decr:
    dec leds
    rcall      shiftOut    ; Sending out the LED data
    rcall      delay250ms ; Wait
    in  read,   PIN        ; Read the switch
    cpi read,  onPos      ; If the read pin is high
    breq     loop        ; If switch state has changed, it will go to loop
    rjmp     decr        ; Otherwise, it will continue to flash
ret

rowLEDs:
    or   leds,    incr      ; Set another LED high while keeping the others on
    rcall      shiftOut    ; Shift out the data
    rcall      delay250ms ; Wait for 1s
    rcall      delay250ms

    in  read,   PIN        ; Read the switch
    cpi read,  onPos      ; If the read pin is high
    brne    anim2       ; Go the other animation if it is low

    lsl  incr
    cpi leds,  0xFF
    brne    rowLEDs     ; Move on to the next LED
    ldi  incr,  1
    clr   leds          ; See if we've reached all 8 LEDs
    rjmp   decr        ; Continue the loop w/o reverting all the variables
    rjmp   decr        ; Revert the increment register
    rjmp   decr        ; Turn off all LEDs (LSB will be turned on)
ret

shiftOut: ; Shift out function
    ldi  mask,  1
    cbi POUT, latch
cycle:
    cbi POUT, clock
    mov  bit,
    and  bit,
    breq  out0
    sbi POUT, data
    rjmp  wrapUp
out0:
    cbi POUT, data
wrapUp:
    sbi POUT, clock
    lsl  mask
    cpi mask, 0
    breq  leave
    rjmp  cycle
leave:
    sbi POUT, latch
ret

delay250ms: ; A delay for 250ms
    ldi  r29, 21
    ldi  r30, 75
    ldi  r31, 191
L1:
    dec  r31
    brne L1
    dec  r30
    brne L1
    dec  r29
    brne L1
    nop
ret

```

Project 22. The 4-Wire DC Fan

Part A. Research

Purpose

This is the beginnings to a four-stage project: programming a four wire PWM fan in assembly. This part will focus on how the PF80251B1-000U-S99 fan is controlled and its operation requirements.

Reference

Fan Datasheet: <http://mail.rsgc.on.ca/~cdarcy/PDFs/Sunon12VDCFanSpec.pdf>

Four Wire PWM Fan Control: <http://mail.rsgc.on.ca/~cdarcy/PDFs/4WirePWMFans.pdf>

Theory

Characteristics

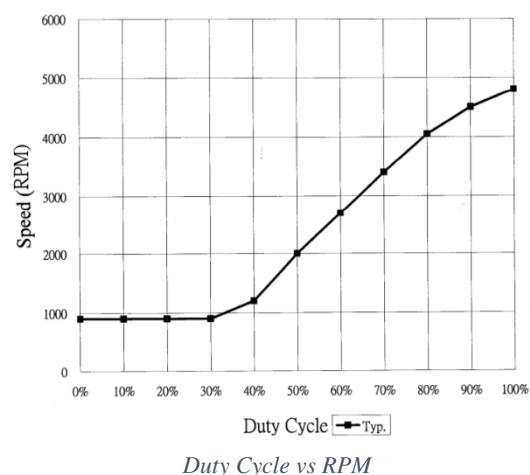
This 2-ball fan operates with a 12V DC power supply, allowing it to rotate at $4800 \pm 10\%$ RPM. This results in a 28.3L/s air flow. The max current the fan will ever draw is 2A. When facing the front of the fan, the blades rotate counter clockwise, causing the air to flow back through the fan.

Wire Functions

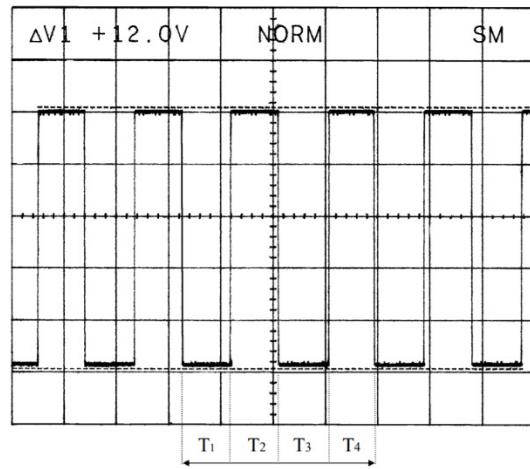
This fan has four wires: 12V power, ground, control, and feedback. The first two wires are relatively simple, sourcing power to the device. To control the fan's motor voltage the control wire is used. This is done by varying the duty cycle of the square wave pulse on this wire; the higher percentage the duty cycle, the more voltage applied to the motor. Specifically, altering the duty cycle from 30% to 100% alters the RPM approximately linearly from 900 to 4800. If the duty cycle is less than 30%, the speed will rest at its 900 RPM minimum.

Wire Colours	
Red	Power (12V)
Black	Ground
Yellow	3 rd Wire (Feedback)
Blue	PWM (Control)

The PWM square wave has very specific requirements in order to work properly. Firstly, the frequency of each cycle must be between 23 kHz and 28 kHz, preferably at 25 kHz. Because of this, the `analogWrite()` function cannot be used, as it only provides a frequency of 980 Hz with the Arduino Uno. Next, a high voltage on the control wire ranges from 2.8V to 6V. The maximum voltage that is considered ground is 0.4V. These voltages are compared with the ground wire. If the applied voltage happens to lie between 0.4V and 2.8V, then the fan will consider it to be floating, defaulting to the maximum RPM.



Sometimes however, there are external factors that affects the RPM in unwanted ways. For this, the feedback wire measures the physical output and reports it to the controller. The feedback wire uses a tachometer output a hall-effect sensor that is mounted on the PCB in the core of the fan. This tachometer creates two pulses per single rotation of the fan (2Hz:1 Rotation). The pulses always have a 50% duty cycle regardless of RPM, so by measuring the amount of time T_1 from the rising edge of a pulse to the falling edge, we can calculate how long it will take for the fan to complete one rotation. This is represented by the equation $RPM = 4 \cdot T_1 \div 60$. If the RPM is far of its expected value, the fan's built-in protection will shut off the motor until the resistance is reduced. This feedback wire is open collector, so a pullup resistor must be attached to prevent floating.

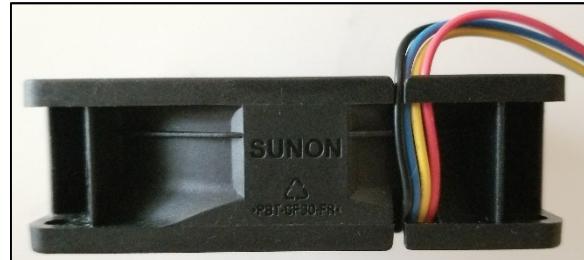


Feedback Wire Pulsing Output

Media



PF80251B1-000U-S99



Wire output of the fan: Ground, PWM, Feedback, & Power

Reflection

This project was not very interesting to me at a cursory glance. However, after researching the part I was surprised by how complex the simplest parts of a computer can be. My most notable reason for liking this fan is that it is a real-life component used in current computers. I even opened my desktop PC to find a four-wire DC fan working the way depicted in the datasheets.

Part B. Execution

Purpose

This project has two core components: a 4-wire fan with speed control and character LCD showing various pieces of data about the current state of the fan. It is all programmed in assembly and wired to an Arduino.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/1819/AssemblyTasks.html#40b>

Waveform Descriptions:

https://www.sparkfun.com/datasheets/Components/SMD/ATMega328.pdf#page=99&zoom=100_0.760

LCD Library: <https://github.com/rsgcaces/AVROptimization/blob/master/LCDLib.asm>

Procedure

The LCD is wired to display the fan data. Since the default pin layout of the LCD library interferes with compare match pins of the timers, the library pins are redefined. The fan is wired to 12 V, ground, OC2B (control), and ICP1 (feedback). While the feedback is not implemented in this project, the potential for it exists. The final component is a potentiometer wired to the first analog read pin of the Arduino.

Parts	Quantity
Arduino (ATmega328p)	1
Four-Wire Fan	1
Character LCD	1
Potentiometer	1

The code first initializes the LCD using Weinman's library and sets up Timer 2, as well as the ADC conversion. Timer 2 is made to have a 25 kHz square wave output on OC2B, controlling the RPM of the fan. Whenever the counter reaches the value of OCR2B, the pin will be cleared and whenever it overflows it will set it. Every time an analog conversion is complete, the eight most significant bits will be read in. This data is then sent to two different places: the LCD and the fan. For this raw reading to be displayed onto the LCD, the data must first be converted into binary encoded decimal, which packs numbers from 0 to 9 into each nibble. This allows for an easy way to convert from binary to ASCII. This is then displayed on the LCD. To send the data to the fan, this eight-bit value must be converted to a percentage of the top value of timer 2. This top value creates the 25 kHz, and the certain value of OCR2B creates a specific duty cycle. The top value is 80, so the reading must be mapped from 0-255 to 0-80. Since $\frac{val}{255} = \frac{OCR2B}{80}$ we can determine that $OCR2B = 80val \ll 8$. This creates incredibly simple arithmetic, as our input is simply multiplied by 80, and then the upper byte is put into OCR2B. This will give the proper duty cycle for the fan.

When it is all put together, the LCD displays the raw potentiometer reading, the duty cycle and theoretical RPM. The fan changes speed as the user turns the potentiometer.

Theory

Timers: Overview

Timers are simply counters that can cause an action, such as an interrupt, when a specific requirement has been met. Typically, this is when the counter overflows. This, by itself, is not too great; a lot of their functionality can be done in software very easily. What makes them great is they run solely using hardware within the microcontroller. This can offload some of the code to the hardware capabilities of the MCU and reduce some of the power required to execute a program.

There are six important registers, five of which need to be adjusted for the timer to function properly. The `TCCR#A`¹ register holds the *Compare Match* (COM) bits and half of the *Waveform Generation Mode* (WGM) bits. On the ATmega328p, there are two COM pins for each timer, named `OC#A` and `OC#B`. By default, these COM bits are cleared, meaning the `OC#A/B` pins are unaffected by the timers. However, by setting these four bits these pins can be toggled, cleared, or set through a timer control software-less interrupt. To tell these pins when to be altered there are two comparison registers, `OCR#A` and `OCR#B`. When the timer has incremented to a point where it is equivalent to one of these registers, it will trigger the compare match interrupt, causing their respective pin to either toggle, set, or clear.

There is another control register, `TCCR#B`. This holds the rest of the WGM bits as well as the prescaler. For the ATmega328p, Timer 0 and 2 have eight different waveforms whereas Timer 1 has 16. Since [waveform generation](#) is a complex topic, there is a dedicated theory section to it. Prescalers on the other hand, are relatively simple. Without a prescaler, every timer will increment once per clock cycle. For the Arduino UNO, this is 16 million times a second. To slow this down, prescalers act as mini timers, incrementing a set number of times before telling the main timer to increment. This, again, is purely done in hardware and its value is controlled by the Clock Select bits in `TCCR#B`.

There are three interrupts associated with each timer. First, there is the Timer overflow interrupt. When the timer reaches its maximum value, it will roll over to zero and cause this interrupt. The next to, as previously mentioned, are the comparison interrupts. When the timer register is equal to `OCR#A/B`, these interrupts will trigger. In order to use any of these interrupts their respective enable bit must be set. All of these enable bits are in `TIMSK#`.

The final register is the `TCNT#` register. This is the place where the incrementing value is stored. Depending on the timer and the MCU, this can be a one- or two-byte register. Because of this, `OCR#A` and `OCR#B` can also be full words instead of a single byte, as depicted in the table.

	7	6	5	4	3	2	1	0
<code>TCCR#A</code>	COM#A1	COM#A0	COM#B1	COM#B0	–	–	WGM#1	WGM#0
<code>TCCR#B</code>	DOT ²	DOT ²	–	/WGM13	WGM#2	CS#2	CS#1	CS#0
<code>OCR#A</code>	Comparison Register(s)							
<code>OCR#B</code>	Comparison Register(s)							
<code>TIMSK#</code>	–	–	–	–	–	OCIE#B	OCIE#A	TOIE#

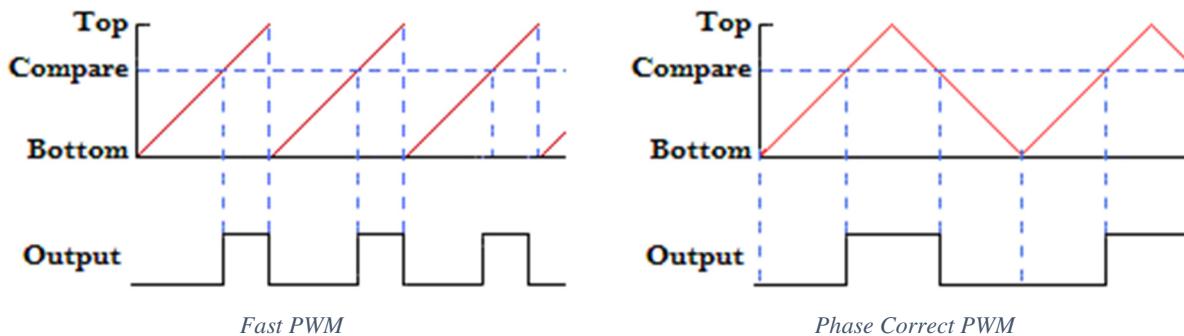
¹ # represents the timer in focus. For example, `TCCR1A` is the Timer Control Register for Timer 1.

² Depends on Timer: Bits are not reserved but are irrelevant for this project.

Waveforms

Timers can increment in two main ways: Fast PWM and Phase Correct (PC) PWM. When Fast PWM is used, the counter will increment to the top value and reset back to zero. CTC and Normal modes also use fast PWM but are named differently in the datasheet for their typical usage. PC PWM begins to decrement once it reaches the top value. For both incrementing patterns, by adjusting the top value (or prescalers) the frequency can be adjusted. This is because the counter will reach the top sooner if the limit is lower. When the comparison register is adjusted, the duty cycle is altered in a predictable pattern. For flexibility with frequency, modes with OCRA as the top should be chosen.

When the COM bits of TCCR#A are used in conjunction with different waveforms, the output of the OCRA/B pins behave differently. For some, when the timer has a value of zero, OCRA/B would be cleared. Others, it will toggle, or be set, or be ignored. To view the specifics, see the ATmega328p datasheet.



MODE	TYPE	TOP	UPDATE OCRA/B AT:	OVF INT. FLAG SET AT:
0	Normal	MAX	Equivalence	MAX
1	Phase Correct PWM	MAX	MAX	0
2	CTC	OCRA	Equivalence	MAX
3	Fast PWM	MAX	0	MAX
4	Reserved	-	-	-
5	Phase Correct PWM	OCRA	MAX	0
6	Reserved	-	-	-
7	Fast PWM	OCRA	0	MAX

Timers: Application

Only one of the three timers are used in this project, with the other two having the potential to be used as well. To control the fan, timer 2 is put on waveform mode seven and COM2A1 bit is set. This clears OC2B on compare match, sets it when it overflows. The overflow limit of this waveform is OCR2A. This allows for control of the frequency and duty cycle with variations in OCR2A and OCR2B respectively. The fan requires a 25 kHz frequency, while the RPM is controlled by the duty cycle. The other timers can be used to schedule LCD updates as well as input capture, but these have not been implemented.

Analog Reading

The ATmega328p uses a 10-bit Analog to Digital Converter. To achieve this, a series of OP-Amp comparators checks if the input voltage is above or below half of the reference voltage (RFV). If it is, it sets the most significant bit of the ADC register and then subtracts half of the RFV. It then repeats this process, comparing the subtracted input voltage to a further division of the RFV, until 10 bits of precision has been reached.

To complete this process, multiple registers must be set up. First, the ADMUX register is used to determine which pin is performing the read. Also, within this register is the ADLAR bit, which determines whether the most significant eight bits are grouped together in the upper byte or right aligned like normal. Next, there are two control registers, ADCSRA and ADCSRB. These hold important enable bits, prescalers and flags. The ADC clock speed should be reduced to 125 kHz to maintain accuracy using the ADPS# bits. To trigger an analog reading, the ADC Auto Trigger Enable (ADATE) bit is set. This causes the ADC to be triggered by an external event depending on the ADTS# bits in control register B. For continuous conversions, these three bits are cleared. To enable the interrupt, the ADC Interrupt Enable bit is set. The final register is DIDR0 which allows each of the analog pins digital use be disabled in order to reduce power consumption.

To ensure proper conversion, the ADC should be run once to ‘warm’ it up. The first conversion tends to be inaccurate and should be avoided. Once the user would like to read the ADC value, the 10-bit data will be stored in ADCH and ADCL.

	7	6	5	4	3	2	1	0
ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
ADCSRB	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0
DIDR0	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D

LCD

Most character LCDs have a backpack on them. In this case, the driver chip used in the backpack is the HD44780U. This chip is a generic driver IC that can be used for Character LCDs of up to 64×2. Although the specifics of controlling the LCD are masked behind a library, the general concepts are clear. The driver has to 8-bit registers that it uses to control the LCD: the instruction register (IR), and the data register (DR). First, the cursor must be set, telling the LCD which segment to write on. To do this, the IR is set to *Cursor Shift*. As the name suggests, this shifts the cursor over to the next character slot. Then the IR is set to *Set DDRAM Address*. This chooses the character from DDRAM, with a *Read Data* putting it into the DR. Then, when the LCD is ready, it displays the data register onto the screen at the specified location.

Code

```

;PROGRAM      :LCDfromADCinClass
;PURPOSE     :Obtains ADC of POT readings and configures it for LCD presentation
;AUTHOR       :C. D'Arcy
;DATE         :2019-04-30
;DEVICE        :Arduino (ATmega328p)
;NOTES        :DD stands for Double Dabble (Shift/Add3) algorithm that converts binary
(8 or 16) to packed BCD (3 or 5)
#include      <prescalers.inc>
#define        TOP          80           ;16MHz/8/25000 = 80 = TOP (OCR2A)
.equ         cTHREE       =0x03   ;DD: CONSTANTS for Double Dabble algorithm
.equ         cTHREEZERO  =0x30   ;DD: ditto
.equ         POTAddress   =0x00   ;LCD cursor addresses for labels
.equ         DCAddress    =0x08   ;
.equ         RPMAddress   =0x40   ;
.equ         TACHAddress  =0x48   ;
.def         index        =r14   ;used as a Z register address offset into the arrays
.def         temp          =r15   ;temporary usage
.def         util          =r16   ;
.def         count         =r17   ;countdown support for various activities
.def         bin0          =r18   ;DD: binary LOW byte
.def         bin1          =r19   ;DD: binary HIGH byte
.def         BCD10         =r20   ;DD: BCD: 2 Least significant BC Digits
.def         BCD32         =r21   ;DD: BCD: Middle BC Digits
.def         BCD4          =r22   ;DD: BCD: 1 Most Significant BC Digit
.def         three          =r23   ;DD Registers: assigned the constant 0x03
.def         threeZero     =r24   ;DD Registers: assign the constant 0x30
; ***** INTERRUPT VECTORS *****
.org        0x0000           ;start of vector jump table
        rjmp   reset            ;lowest interrupt address => highest priority!
.org        ADCCaddr        ;
        rjmp   ADC_vect         ;
.org        INT_VECTORS_SIZE
; ***** LCD STRING LABELS *****
POTLabel:
.db         "POT:",0,0           ;must be null-terminated (with even padding)
DCLabel:
.db         "DUTY:",0,0           ;
RPMLabel:
.db         "RPM:",0,0           ;
TACHLabel:
.db         "TCH:",0,0           ;
DUTYTable:
        ;p.11:http://mail.rsgc.on.ca/~cdarcy/PDFs/Sunon12VDCFanSpec.pdf
.db         " 010203040506070809010"
RPMTTable:
        ;p.11:http://mail.rsgc.on.ca/~cdarcy/PDFs/Sunon12VDCFanSpec.pdf
.db         " 900 900 900 9001200200027003400405045004800"

reset:
        ldi    util, low(RAMEND)    ;PC jumps to here on reset interrupt...
        out   SPL, util            ;initialize the stack pointer
        ldi    util, high(RAMEND)   ;
        out   SPH, util            ;
        ldi    three, cTHREE        ;DD Support

```

```

ldi    threeZero,cTHREEZERO      ;      "
call   ADCSetup                ;Configure the ADC peripheral
call   LCD_init                ;initialize devices and MCU peripherals
call   T2Setup
sei
wait:
rjmp  wait                   ;repeat or hold...
ret
;unreachable

ADCSetup:
ldi    util, 1<MUX0          ;turn off digital use of A0 tp save power
sts   DIDR0, util            ;disable digital use of A0 (pin 14)
ldi    util, 1<<ADEN         ;enable the A/D peripheral
sts   ADCSRA,util           ;
ori   util, 1<<ADPS2 | 1<<ADPS1 | 1<<ADPS0    ;divide down AD clock to 125kHz
(recommended for accuracy)
sts   ADCSRA,util           ;set it

clr   util                   ;Ignore the ACME bit
sts   ADCSRB,util           ;free running, for now

ldi   util, 1<<REFS0 | 1<<ADLAR ;AVCC reference (5V) and Left Adjust
sts   ADMUX, util            ;
ldi   util, ADCSRA          ;prepare for a dummy conversion (recommended)
ori   util, 1<<ADSC          ;set the SC flag
sts   ADCSRA,util           ;tstart your covnversion

dummy:
lds   util, ADCSRA          ;read the register
sbrc util, ADSC              ;
rjmp dummy                  ;
lds   util, ADCSRA          ;
ori   util, 1<<ADSC|1<<ADIE ;Start first conversion and enable interrupt
sts   ADCSRA,util           ;
ret

T2Setup:
sbi   DDRD, PD3              ;PWM on pin 3 (OC2B)
ldi   util, 1<<COM2B1 | 1<<WGM21 | 1<<WGM20
sts   TCCR2A,util            ;
ldi   util, 1<<WGM22 | T2ps8
sts   TCCR2B,util            ;
ldi   util, TOP               ;
sts   OCR2A, util            ;
clr   util                   ;
sts   OCR2B, util            ;
ldi   util, OCIE2B            ;
sts   TIMSK2,util             ;
ret

ADC_vect:
lds   temp, ADCH              ;Grab the most significant 8 bits
mov   util, temp              ;let's determine the duty cycle...
clr   r17                     ;
ldi   r18, TOP                ;

```

```

    clr    r19                      ;
    mul    r18,  util                ;
    sts    OCR2B, r1                 ;set the duty cycle

    clr    index
    mov    util,  temp
again:
    subi   util,  25
    brlo   continue
    inc    index
    rjmp   again
continue:
;perform the Double Dabble algorithm on the ADC value
    mov    bin0,  temp              ;restore ADCH value and prepare for DD
    clr    bin1                    ;prepare bin1 (bin0 is already to go)
    call   bin16BCD               ;convert to BCD
    call   LCDUpdate               ;update the LCD
    lds    util,  ADCSRA            ;start the next conversion
    ori    util,  1<<ADSC          ;
    sts    ADCSRA,util            ;

reti

LCDUpdate:
    ldi    ZH,  high(POTLabel)      ;point to the base address of POT Label
    ldi    ZL,  low(POTLabel)       ;
    ldi    util,  POTAddress        ;set the target address on the LCD
    call   lcd_write_string_4d     ;display it
    mov    util,  BCD32             ;obtain the hundreds & thousands digits
    andi   util,  0x0F              ;we only want BCD2 (low nibble)
    ori    util,  48                ;add 48 to get the ASCII value
    call   LCDWriteCharacter       ;write it
    mov    util,  BCD10             ;obtain the units and tens digits
    swap   util                   ;work with the tens digit first

    andi   util,  0x0F              ;mask it
    ori    util,  48                ;add 48 to get the ASCII value
    call   LCDWriteCharacter       ;write it
    mov    util,  BCD10             ;reload
    andi   util,  0x0F              ;mask off the tens digit
    ori    util,  48                ;add 48 to get the ASCII value
    call   LCDWriteCharacter       ;write it

    ldi    ZH,  high(DCLabel)       ;
    ldi    ZL,  low(DCLabel)        ;
    ldi    util,  DCAddress         ;
    call   lcd_write_string_4d     ;
lsl    index                         DisplayDuty
    call   ZH,  high(RPMLabel)      ;
    ldi    ZL,  low(RPMLabel)       ;
    ldi    util,  RPMAAddress        ;
    call   lcd_write_string_4d     ;

lsl    index                         DisplayRPM
    ldi    ZH,  high(TACHLabel)     ;

```

```

        ldi    ZL,    low(TACHLabel)
        ldi    util,   TACHAddress
        call   lcd_write_string_4d
ret

LCDWriteCharacter:
        call   lcd_write_character_4d      ;display the character
        ldi    util,   80                 ;40uS delay (min)
        call   delayTx1uS
ret

DisplayDuty:
        ldi    ZH,    high(DUTYTable)<<1
        ldi    ZL,    low(DUTYTable)<<1
        add   ZL,    index

        lpm   util,   Z+
        call   LCDWriteCharacter
        lpm   util,   Z+
        call   LCDWriteCharacter
        ldi    util,   '%'
        mov   count, index
        cpi   count, 20
        brne  bypass
        ldi    util,   '0'

bypass:
        call   LCDWriteCharacter
ret

DisplayRPM:
        ldi    ZH,    high(RPMTTable)<<1
        ldi    ZL,    low(RPMTTable)<<1
        add   ZL,    index

        lpm   util,   Z+
        call   LCDWriteCharacter
        lpm   util,   Z+
        call   LCDWriteCharacter
        lpm   util,   Z+
        call   LCDWriteCharaacter
        lpm   util,   Z+
        call   LCDWriteCharacter
ret

; **** End of Main Program Code ****

#define          LOCALLCD           ;This allows user code to override default
(Appliance) wiring
.equ   lcd_D7_port      = PORTB           ;lcd D7 connection
.equ   lcd_D7_bit       = PORTB4
.equ   lcd_D7_ddr       = DDRB

.equ   lcd_D6_port      = PORTB           ;lcd D6 connection
.equ   lcd_D6_bit       = PORTB3
.equ   lcd_D6_ddr       = DDRB

```

```
.equ lcd_D5_port = PORTB ;lcd D5 connection
.equ lcd_D5_bit = PORTB2
.equ lcd_D5_ddr = DDRB

.equ lcd_D4_port = PORTB ;lcd D4 connection
.equ lcd_D4_bit = PORTB1
.equ lcd_D4_ddr = DDRB

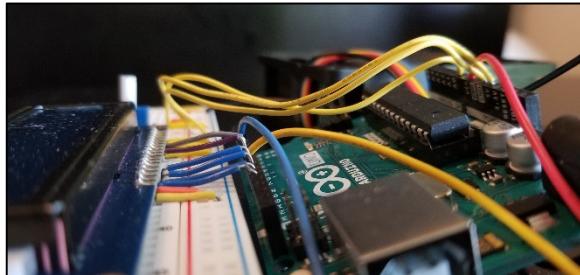
.equ lcd_E_port = PORTD ;lcd Enable pin
.equ lcd_E_bit = PORTD7
.equ lcd_E_ddr = DDRD

.equ lcd_RS_port = PORTD ;lcd Register Select pin
.equ lcd_RS_bit = PORTD6
.equ lcd_RS_ddr = DDRD

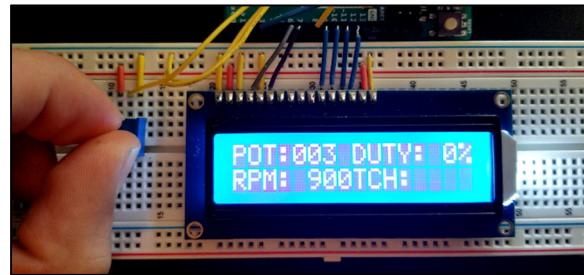
#include <LCDLib.asm> ;Weinman's LCD library code
#include <DoubleDabble.asm>
```

Media

Personal Video: <https://youtu.be/ec6GigbML2w>



Wiring Close Up



LCD in Action

Reflection

This project was intense, but I feel as though I understood. It is kind of shame that it has been squeezed into a busy time with so many other school projects, but I'm glad we did it. However, I think the most enjoyable part for me was learning about the LCD. I'd love to further my knowledge of character and graphic LCDs in university.

Project 23. Medium ISP: Sound Controlled Power Bar

Purpose

This project initially started off as a voice recognition module that could tell the time and control a remote power bar. As it evolved however, the emphasis was placed on the power bar itself, and controlling it remotely. The final product has two components: a recognition module and a control module. The recognition module uses radio frequencies to tell a PCB when a clap occurs. This PCB then alters the outlets on a relay-controlled power bar. The PCB uses surface mount technology (SMT) and is encased in a slim 3D printed housing.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/1819/ISPs.html>

Voice Control: https://cdn.sparkfun.com/datasheets/Sensors/Sound/EasyVR_Datasheet_2.3.pdf

Regulator: <https://www.analog.com/media/en/technical-documentation/data-sheets/3638fa.pdf>

Procedure

To begin, two prototypes are made; one to control the power bar, and another to tell it when a sound has been recognized. The former has a PCB made with an NRF, MCU, and a voltage regulator, along with their concurrent parts. This allows it to receive data, interpret it, and control the outlets accordingly.

A large component of this project is finding an efficient way to source 5 V. Since the PCB is mounted onto a power bar, there are plenty of options. A USB could be added to directly plug into one of the outlets, but this doesn't provide a satisfying user experience as it takes some of the utility away. Instead, the decision is made to source power from the internal PCB of the

power bar. Various AC and DC power sources are found with different voltages levels, but in the end a switching regulator, sourced from 100 VDC, is used to generate the PCB's 5 V.

Initially, a voice control module is purchased to perform the recognition, lessening the load on the Arduino. However, the device was not able to function in time for this ISP. Another potential solution is to use the Üspeech Arduino library, but it is too cumbersome and imprecise to be useful. Instead, the [MSGEQ7](#) chip is used to detect claps. By only analyzing the amplitude of high frequencies, a clap can be detected. Once this occurs, a signal is sent through the NRFS to the PCB. Unfortunately, the pitch for the FFC converter was incorrectly chosen. This means that code is not able to be uploaded to the onboard ATmega328p.

Parts	Quantity
ATmega328p	2
MSGEQ7	1
Electret Microphone	1
NRF24L01	2
LTC3638 Voltage Regulator	1
1 μ F Capacitor	2
22 pF Capacitor	2
10K Ω Resistor	1
2K Ω Resistor	1
1K Ω Resistor	1
220 μ H Inductor	1
16 MHz Crystal	2
Schottky Diode	1
1 mm Pitch FFC Converter	1

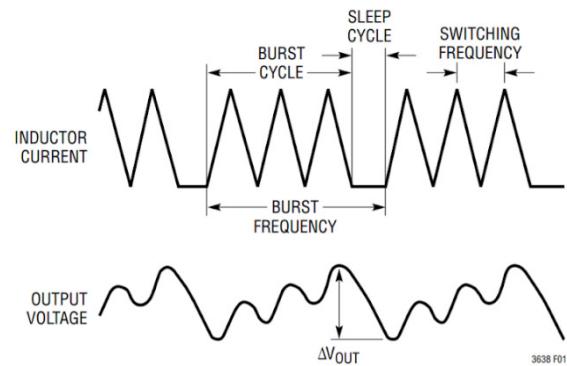
Theory

Audio Recognition

To recognize audio, there are several different techniques, all with varying capabilities and challenges. Devices that can detect phrases generally compare the audio input with five-to-seven pre-recorded messages. If there is a comparison with over a 90% similarity, it will trigger an output than can be read by the main microcontroller. This signal can come in many forms, such as I₂C, SPI, UART, or simply GPIO. Certain techniques, such as the Üspeech library or the MSGEQ7 circuit detect frequencies and can be used to decipher consonants.

Step-Down Converters

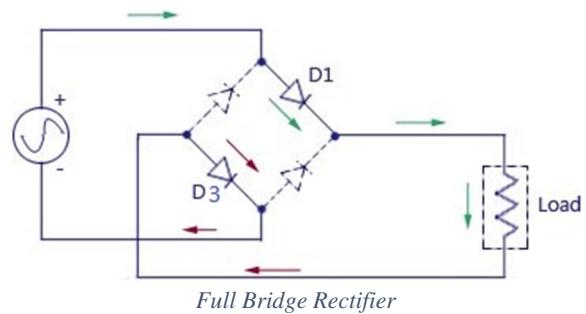
A step-down converter (also known as a buck converter) is a type of switching regulator that decreases a DC input voltage efficiently. Instead of converting the unwanted power into heat, it uses a pulsing technique, similar to PWM, to decrease the voltage. To achieve a proper output there are two components: a burst and a sleep cycle. The burst cycle rapidly increases and decrease the output current, creating a triangle wave. The frequency of this wave is determined by the inductance and capacitance of external components. This cycle continues until the feedback pin, V_{FB}, detects the peak expected voltage. Once this occurs, the output moves onto the sleep cycle. During this phase, the main source of power is from the external capacitor and inductor. Once the output voltage drops below 0.8 V, the feedback pin resets the entire process. When the LTC3638 decreases the voltage from 140 V to 5 V, the expected efficiency is 76%.



Alternating Current

Alternating current, as opposed to direct current, switches the current back and forth between two wires. This has two main uses: motors and voltage reduction. While most hobby motors are brushed DC motors, they essential convert the power to AC. This is because as the motor rotates, the brush connects to a different plate, switching the current. When using an AC motor, this doesn't need to be a part of the interior, so as a result they are more compact and efficient. The main reason AC is used in power lines is because of the high voltage that is used in them. When this high voltage reaches a house, it must be reduced. Historically, DC has not been able to provide efficient methods of power reduction. AC, on the other hand, can use transformers to drop voltage levels with a 99% efficiency! This is the only reason AC is used in North American power grids; lots of the United Kingdom simply uses high voltage DC (HVDC), as new methods for decreasing DC voltage are being introduced.

To convert AC to DC, a full bridge rectifier can be used. As the current alternates, the forward diodes take turns providing the source for current, and the back diodes provide a path to ground. A capacitor can be used between DC power and ground for smoothing.

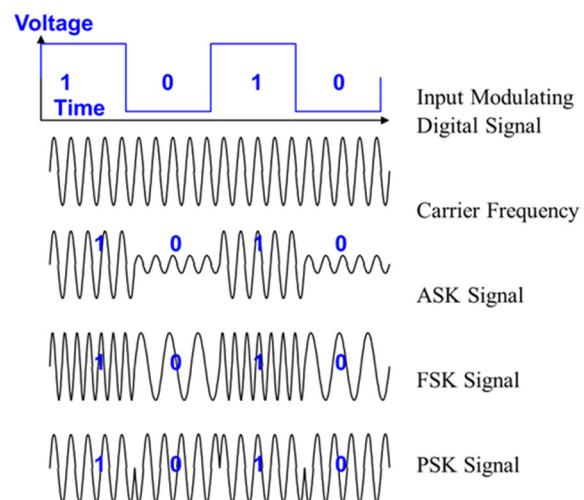


NRFs

The two NRFs in the project are used to communicate information from the recognition module to the control module. NRFs, like most forms of wireless communication, uses radio frequencies to transmit data. To distinguish it from other techniques, it uses a 2.4 GHz – 2.525 GHz frequency range.

There are three different variables to account for when using an NRF: data rate, bandwidth, and channel frequency. The data rate is analogous to the clock cycle; It tells the NRF how quickly to move to the next bit when transmitting or receiving data. The options for the NRF24L01 are 250 kb/s, 1 Mb/s, and 2 Mb/s. At first glance the higher bit rate seems like the preferred option, but it comes with a cost. As the data rate increases, so does the bandwidth. The bandwidth is the range of acceptable frequencies an NRF can output, before being ignored by the receiver. When choosing a bit rate, both factors must be considered. Finally, there is the channel frequency. To make two NRFs communicate with each other, they must be on the same frequency range.

Radio frequencies can typically transmit data in two different ways: AM (ASK) or FM (FSK). AM, or amplitude modulation, varies the amplitude of the signal being transmitted. A high amplitude represents a high bit and a smaller amplitude represents a low bit. FM, or frequency modulation, varies the frequency (as the name suggests). In this case, a high frequency represents a high bit. NRFs use a specific type of FM known as Gaussian Frequency Shift Keying (GFSK). GFSK is a simple form of FM radio frequencies; it has no representation for a new bit (such as PSK), but it does provide a smoothing between bits. This increases the likelihood of having a bit be mistaken for another, but reduces interference with neighbouring channels, allowing for a decreased bandwidth.



Reflection

This project has evolved so much since its beginning. The simple decision of where to source the PCB's power changed the focus of the ISP in a way I never anticipated. Learning about AC, inductors, and voltage regulation was always something that provoked my curiosity, but I never thought it would work its way into a project. I'm disappointed the voice recognition failed so early in the process, but it has finally taught me a lesson that has taken me two years to learn; I enjoy delving into the complexities of known parts more than I like debugging new ones. My next ISP will hopefully reflect this.

Code

```
// PROJECT: NRF Transmitter
// DEVICE: ATmega328p
// AUTHOR: Daniel Raymond
// DATE: 2019-02-05

#include <RF24.h>

// Global variable setup here
#define strb A1
#define reset A0
#define inPin A2
#define CE 7
#define CSN 8
#define WAIT 2000

RF24 radio(CE, CSN);
const byte address[6] = "00001"; // Channel Freq = 2400 + address [MHz]
uint8_t data = 0;

uint16_t freq;
long tmDelay = millis();
boolean allowRead = false;

void setup() {
    DDRC = 0x06; // Sets up pin
    DDRD = 1<<PD5;

    pinMode(CSN, OUTPUT);
    pinMode(CE, OUTPUT);
    _SPI.begin(); // Sets up the NRF
    digitalWrite(CE, LOW);
    digitalWrite(CSN, HIGH);
    radio.openWritingPipe(address); // Give channel to communicate on
    radio.setPALevel(RF24_PA_MIN); // Set Power Amplifier Level
    radio.stopListening(); // Set to be the transmitter
}

void loop() { // Loop that makes the cycle continue forever
    // Reset input for the MSG
    digitalWrite(reset, HIGH);
    digitalWrite(reset, LOW);

    for (uint8_t i = 0; i < 7; i++) {
        digitalWrite(strb, LOW); // Sets MSG latch to be able to read in
        if (i == 6)
            freq = analogRead(inPin); // Only reads the high frequency
        digitalWrite(strb, HIGH); // Moves on to the next frequency
    }

    if (freq >= 30 && allowRead) {
        radio.write(&data, sizeof(data)); // Write the data to the NRF
        tmDelay = millis();
        allowRead = false;
    }

    if (millis()-tmDelay > WAIT)
        allowRead = true;
}
```

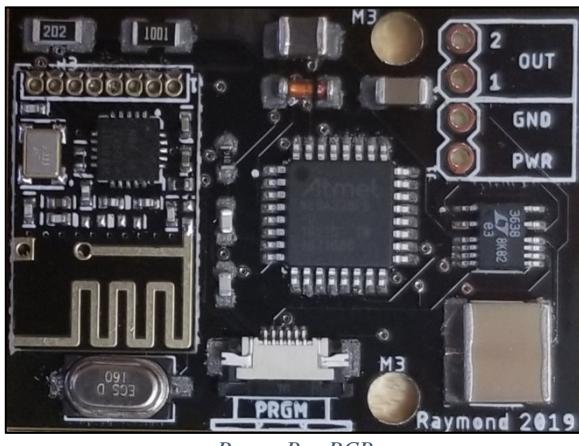
```
// NRF Receiver
#include <RF24.h>

#define CE 7
#define CSN 8
RF24 radio(CE, CSN);
const byte address[6] = "00001";
uint8_t data;

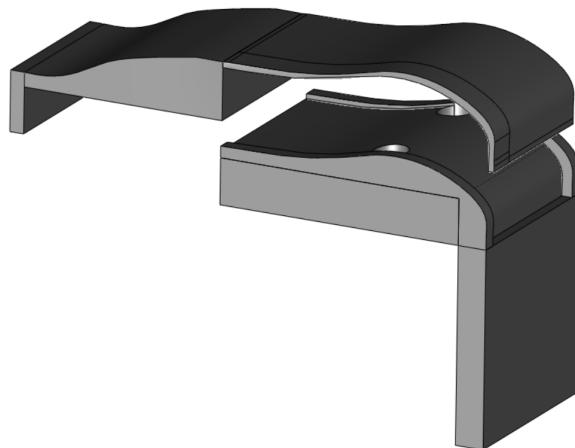
void setup() {
    radio.begin();
    radio.openReadingPipe(0, address); // Give channel to communicate on
    radio.setPALevel(RF24_PA_MIN); // Set PA level
    radio.startListening(); // Set to be receiver
    DDRD = 0x0F; // Set direction of OUTPUT Pins
}
void loop() {
    if (radio.available()) {
        radio.read(&data, sizeof(data)); // Read in the data when it is available
        PORTD = data; // Output the values onto the power bar
    }
}
```

Media

Personal Video: <https://youtu.be/b-euUTvDf28>



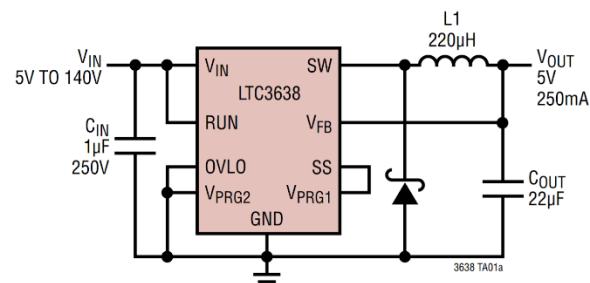
Power Bar PCB



2-Piece Case for the PCB



Relay Controlled Power Bar (IoT Relay 2)



Buck Converter Schematic

Project 24. Long ISP: CHUMP Extended

Purpose

This project is a continuation of the 4-bit programmable computer built in September. Numerous components have been added to the design, including a Program Counter (PC) extension, LED matrix, and a coding input system (CIS). This is a partner project with James Corley; he primarily focused on the input and case whereas I focused on the output and the PCB.

Reference

Project Page: <http://darcy.rsgc.on.ca/ACES/TEI4M/1819/ISPs.html>

PC Ext. Datasheet: <http://www.ti.com/lit/ds/symlink/sn74ls93.pdf>

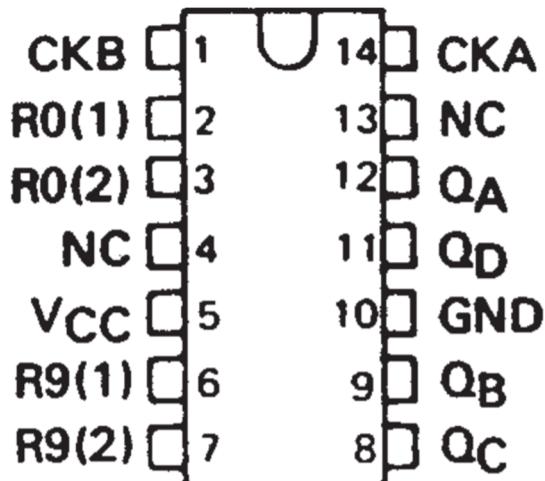
Theory

PC Extender

Since the original CHUMP had a four-bit PC, only 16 total lines of code could be used.

Considering that two daisy-chained shift registers requires a minimum of 33 lines, this isn't practical. Instead of completely overhauling the current PC, a four-bit counting IC is used to extend it, allowing for 256 possible lines of code.

The chip used is the *SN74LS93N*. It has three core components: a clock input, reset, and a four-bit output. This chip takes in the falling edge of a pulse, incrementing its output by one. Since we only want these bits to increment when the PC overflows, the MSB of the PC becomes the clock input. There is no load functionality to the IC, but it can be cleared by two reset pins; when both are high, the output will be cleared immediately. When performing a jump, this reset pin must be pulled high. This allows jumps anywhere from 0 to fifteen. However, there is one subtle problem. When the PC load is grounded, it will load in a new four-bit value on the next clock cycle, whereas this IC will clear itself immediately. To work around this, the reset is first sent through the extra flip-flop of the address register. The flip-flop will store the data, resetting it on the next clock cycle.



Output

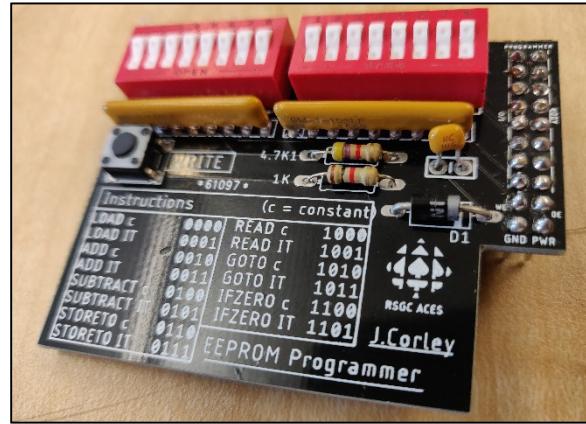
The output of the computer is the three LSBs of the accumulator. In ascending order (0→2), the pins tied to the accumulator are the latch, data, and then the clock. To control the 8x8 LED matrix on the board, there are two daisy chained shift registers. Their latch and clock pins are tied together, reducing the total number pins required to control them down to three. To transmit data, the latch is grounded, and the clock pin is pulsed. This will cause the shift register to read in the data line. This clock signal is pulsed 16 times: enough to fill both registers. Then, the latch is pulled high, outputting the data onto the shift registers. Unfortunately, outputting a value from RAM to the ALU will affect the pins of the shift registers, so using RAM is not viable.

CIS

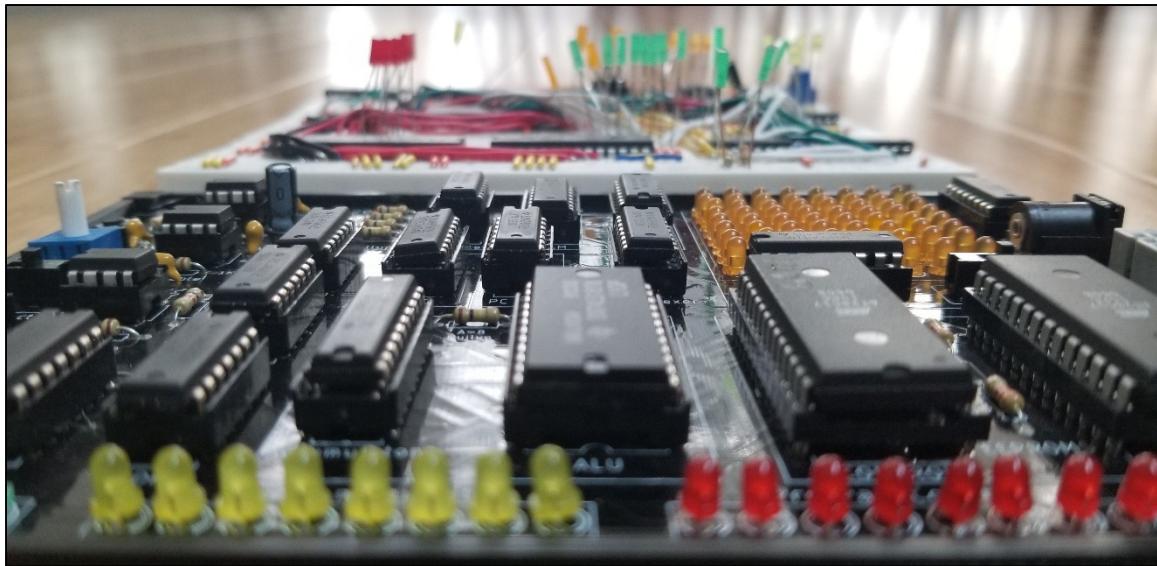
Although James is the head engineer for the CIS, it is important that everyone knows a broad overview of every component. The EEPROM that contains the code is the *AT28C17* EEPROM. To write to it there are two control pins, as well as a byte of address pins and a byte of data pins. The control pins are Write Enable (WE) and Output Enable (OE); they are both active-low. On the central PCB WE is pulled high and OE is pulled low. This sets the EEPROM into output mode, where it can be used by the computer. To program it, OE is pulled high. The two switch banks can be used to control the state of each bit.

Although this EEPROM has 1024 bytes of possible storage the computer is only able to access 256 of them.

Once the 8/10 address pins determine what line of memory the byte is being stored in, the data pins can be toggled to control the byte that is being inserted. When all the appropriate switches have been made, WE is pulsed, flashing that specific line of code into the EEPROM.



Programmer PCB



PCB & Breadboard Together

Operation Codes

In this project we have continued to use the same [operation codes](#) as in the breadboard version, with one small edit. As a brief refresher, the upper nibble of each program line details which instruction the computer will execute while the lower nibble acts as a constant. The operation code (Op Code) is given to the control EEPROM, which determines the state of certain chips.

The change is because a logical *AND* operation has been added. However, adding this instruction to the end of the instruction table was not viable. To determine whether the PC should be loaded to, the middle bits of the Op Code were OR'd and then that result was AND'd with the MSB. This told the computer that it was doing either a *GOTO* or an *IFZERO*, both of which had the possibility of the jumping. This was then NAND'd with the *A=B* pin of the ALU to allow for the *IFZERO* to function. This was fine for the breadboard, but with the introduction of a new operation, the system breaks. If the new operation is performed and the result happens to be 15, then the computer would perform a *GOTO*. A solution to this would be to XOR the middle bits instead of simply ORing them, but this would require massive restructuring of the device.

Instead, the jump commands are shifted down in the instruction table. This changes the resulting logic, but it is simpler than the original. To determine if there is a possibility of jumping, the two MSBs are AND'd together. The slot where the *GOTO* used to be is then filled up by the new instruction and won't trigger any accidental jumps.

Instruction	Op Codes	ALU	S ₃	S ₂	S ₁	S ₀	M	Cn	Acc	R/W
LOAD const	0000	B	1	0	1	0	1	X	0	X
LOAD IT	0001	B	1	0	1	0	1	X	0	1
ADD const	0010	A PLUS B	1	0	0	1	0	1	0	X
ADD IT	0011	A PLUS B	1	0	0	1	0	1	0	1
SUBTRACT	0100	A MINUS B	0	1	1	0	0	0	0	X
SUBTRACT IT	0101	A MINUS B	0	1	1	0	0	0	0	1
STORETO	0110	ignore	X	X	X	X	X	X	1	0
STORETO IT	0111	ignore	X	X	X	X	X	X	1	0
READ const	1000	ignore	X	X	X	X	X	X	1	1
READ IT	1001	ignore	X	X	X	X	X	X	1	1
AND const	1010	AND	1	0	1	1	1	X	0	X
AND IT	1011	AND	1	0	1	1	1	X	0	1
GOTO const	1100	Logic 1	1	1	0	0	1	X	1	X
GOTO IT	1101	Logic 1	1	1	0	0	1	X	1	X
IFZERO const	1110	Not A	0	0	0	0	1	X	1	X
IFZERO IT	1111	Not A	0	0	0	0	1	X	1	X

Procedure

This is the most complex PCB designed in this entire Design Engineering Report. However, the working breadboard version is used as a reference to make the schematic creation process much easier. Each component of the CHUMP is designed discretely and wired together after thorough checking. Since there are some unusual parts used, an order is placed on Digikey for chip seats, multiplexors, headers, and other parts the were low in stock. The CIS PCB is designed to attach to the central PCB via a 20-pin header. This allows for communication of the address, data, control pins, as well as supplying power and ground to the external board.

Once the board arrives, the clock is soldered onto the PCB. This is where the first problem was encountered: since the clock was designed separately, there is a different naming convention for the power. This is fixed using a simple wire, connecting the power traces. Then the program counter and its eight LEDs are soldered, working as intended. Next, the EEPROMs seats are soldered and tested. They are functional, but the CIS programmer short circuits the computer. This is because its output is tied to the PC output. When they output different signals, a short occurs. To fix this, an eight-bit multiplexor is added to decide between the two signals. Since the rest of the board is incredibly interconnected, the entire board is soldered up except for the shift registers and the LED matrix. The third problem: the multiplexor input that is connected to the program ROM was improperly labeled. On the ROM the pins were labeled from zero to three, whereas the multiplexor was labelled from one to four. While not detrimental, this is considered when performing load commands. The ROM is programmed to match the code of the breadboard, which works flawlessly. For a comprehensive parts list, see the CHUMP.

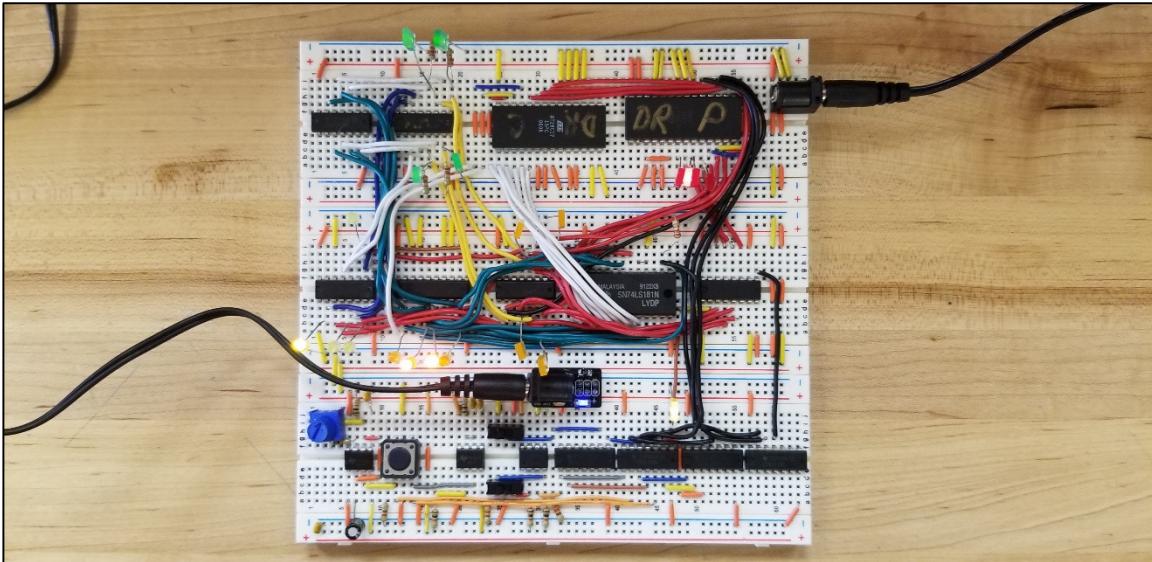
Code

Since this is a programmable computer, the code can literally be anything. Below is simply a way of turning on all the LEDs. To see the code that is running in the video, see [project 17](#).

Line	Ground Data		Line	Power Data	
0	LOAD	0000	17	LOAD	0110
1	LOAD	0100	18	LOAD	0010
2	LOAD	0000	19	LOAD	0110
3	LOAD	0100	20	LOAD	0010
4	LOAD	0000	21	LOAD	0110
5	LOAD	0100	22	LOAD	0010
6	LOAD	0000	23	LOAD	0110
7	LOAD	0100	24	LOAD	0010
8	LOAD	0000	25	LOAD	0110
9	LOAD	0100	26	LOAD	0010
10	LOAD	0000	27	LOAD	0110
11	LOAD	0100	28	LOAD	0010
12	LOAD	0000	29	LOAD	0110
13	LOAD	0100	30	LOAD	0010
14	LOAD	0000	31	LOAD	0110
15	LOAD	0100	32	LOAD	0001
16	LOAD	0010	33	GOTO	0000

Media

Personal Video: <https://youtu.be/vZ13xud0qBc>
James' Video: <https://youtu.be/C6ym7kKK5XI>



Bread Board



PCB

Reflection

We've gone out with a bang. I loved this project so much and it is nice to have something physical to bring to university and be proud of. Even though the project is officially finished, I'm glad we are bringing it through the finish line and getting version II. It will be a nice conversation starter and reminder of my time in the DES. In grade 9, I dreaded doing hardware, but simultaneously knew that I would go through the entire program. I was destined for it. Thanks to Hardware, I've finally found something I'm passionate about.

Project 25. Flex: Snake Video Game

Purpose

This is the game *Snake* played on an SMT eight by eight LED matrix. The goal of the game is to direct your snake into apples. If you eat one, your snake will grow in length. If you hit the edge of the screen or the body of the snake, the game ends. To execute the code, a surface mount flexible circuit board is made, powered by a flexible solar panel. This is what is laminated into this report.

Reference

Project Page:

darcy.rsgc.on.ca/ACES/TEI4M/1819/Task.html#FP1

Arduino C Code:

<https://github.com/draymond63/Snake>

Procedure

To start, the PCB is designed in Eagle. The circuit is relatively simple; it contains a generic ATmega328p, two shift registers, an LED matrix and some buttons. Since the shipping time is quite lengthy, a board is sent out as soon as possible. It is designed with two pads on opposite ends of the board that are to be connected to the power rails of the solar panel. There is a programming finger designed to fit into a 1mm pitch FFC connector. This is then chained into a laptop to allow for programming.

To begin programming as soon as possible, a prototype is made with through-hole versions of the PCB. Since this is one of my most ambitious AVR assembly projects, the code is first written in Arduino C. Once the entire C code is fully functional, it is fully converted to AVR assembly.

Things such as arrays and variables take quite some time to port over. All the accumulators are used for data manipulation and storage of key variables. Booleans, lesser variables, and arrays are stored in SRAM. Once the PCB and Digikey parts arrive, the circuit is soldered. The code is altered slightly to function on the PCB.

Flex Page

Solar Panel

Theory

Shifting an Array

In SRAM, eight bytes are allocated to holding the status of each LED in the matrix. Each byte corresponds to a different row and each bit in that byte corresponds to a different LED. Two shift registers control the anode and cathode pins of the matrix. To output this data, a full byte is applied to the power register. The index of this byte in the array becomes the row that is grounded. This process cycles through the array, creating the familiar [POV](#) effect.

Parts	Quantity
ATmega328p	1
SNCH595	2
PBNO	4
10K Ω Resistor	5
Diode	14
LED	64

Snake Tracking

Four main pieces of data are kept track of in order to maintain control of the snake's position: the coordinates of head, the coordinates of the tail, the direction the head is going, and the direction the tail is going. Using these pieces, we can ensure the snake is always the correct length and moving in the proper manner.

The head starts off at the top left of the screen, represented by $(0, 0)$. Based on the direction of the head at the given time, one of the coordinates will increment or decrement accordingly. For example, if the head direction is down, then the Y component will increment by one. This coordinate is then set on the LED array and is lit up when the timer is triggered.

The tail's coordinates behave in the exact same manner as the head, but instead of setting a bit in the LED array, it clears it. The tail behaves as an “inverse head”, moving along, turning off LEDs as it goes. To keep determine which direction the tail is going to move, an array of directions is made; it holds all the previous moves that the head has done. This length of this array needs to match the length of the snake. However, the length of the snake is dynamic, so to ensure enough space is allocated for the array, it is given 64 bytes: the amount of space needed to track a snake that has filled the screen. On every timing triggered event, a movement index tells the tail which direction from the array to grab. The tail moves, then the used cell is replaced with the current direction of the head. This ensures that when the snake has moved the length of the snake, the tail will be at the old coordinates and will move in the same direction the head did. When the length of the tail increases, the tail skips a turn and all the data in the directions array above the movement index is shifted up to fill the new available slot.

Timing

The screen is continually refreshing, using POV to display all the proper LEDs. However, timer 1 is used to schedule when the screen updates with new information. Whenever timer 1 overflows it will change a boolean value. On the next cycle of the loop it will alter the LED array that is outputted on the matrix. Since timers overflow incredibly quickly, a prescaler is used. This counts the overflows using hardware, then interrupts the MCU once it has counted the specified amount. The specifics to setting up a timer interrupt are discussed in detail in the [fan](#) project

Collisions

Collisions can occur when the snake's head goes off the screen (hitting a boundary) or trying to cross over its own body. When this happens, the player loses the game. Detecting these occurrences is surprisingly simple. While checking if the coordinates of the snake are greater than 7 would do (since negatives on an unsigned integer result in 0xFF), this would cause a frame where the player has gone off the screen but hasn't lost. This is fine, but easily solved. Instead, the code checks if the player is at a boundary and trying to go past it. To check if the snake is attempting to cross itself, the moveHead function checks to see if the LED it is trying to turn on is already on. If that bit in the array is zero, then it is not going to crossover with itself. The only thing to be wary of is the apple, which sets an LED on that is not a part of the snake. To work around this, the code first checks if the head is going to be on the apple. If it is, it ignores the lose function and sets the appleEaten boolean to true.

Button Detection

There are four buttons paired with each possible direction. Instead of polling each button to see if anything action has occurred, all the inputs have been tied to an external interrupt pin with diodes. This allows the MCU to know instantly when an input has occurred. Then, on the next iteration of the loop, the btnPress boolean will be true, and it runs a function to poll the buttons and change the head direction accordingly. Each direction is tied to a certain bit position, so the raw reading can be put into the headDir register without needing to convert it in any way. To check if this new direction is attempting to backtrack, it is compared with the previous direction using a clever algorithm. If the sum of the new and old directions equal to the sum of two opposing directions, it is attempting to backtrack. For example, $LEFT + RIGHT = 1$. If the sum of the new and old direction also equals one, then the snake must be backtracking. If this happens the new direction is scrapped, and the head direction is reverted.

Apple Spawning

Apples generate at random coordinates. To achieve this, an analog read is performed. This creates a ten-bit conversion value, but since it doesn't matter what the actual reading is, the lower byte is masked to show only the three least significant bits. These bits have the most variability and are stored into the apple's coordinates (operation occurs twice; once for X, once for Y). This spawn function is called once at the beginning, and then whenever the appleEaten boolean is set.

Reflection

This project was lots of fun for me. I've finally learned that I love intense coding challenges. Most of my previous projects have involved complex parts, but this was simply an LED matrix and some buttons; the only challenge was the code. I loved solving the problems in C, and then porting my solutions over into assembly. This is my biggest software undertaking yet: 600 lines of AVR assembly code. Designing circuit boards is a fun, but stressful process. I enjoy it and wish to continue making boards through university. Surface mount PCBs are scary and had me worried that nothing would work. The idea of a flexible circuit is still crazy to me. As of writing this report, it does not work on the PCB, but I honestly think I still have a shot if I solder another (the LED matrix on the current one seems like a fluke to me).

Code

```
; PROJECT:    LED Snake
; PURPOSE:   The code for the flex PCB
; DEVICE:    ATmega328p
; AUTHOR:   Daniel Raymond
; DATE:     2019-04-15

#include <prescalers.inc>
#include "define.inc"

.org      0x0000      ; Tells PC that reset is the 1st instruction
    rjmp  reset
.org      INT0addr
    rjmp  INT0_vect
.org      OVF1addr
    rjmp  TIMER1_OVF_vect

.dseg
.org baseArray
LEDs:
    .byte  matrixWidth ; Reserves eight bits

.org (baseArray + matrixWidth)
snakeMove:
    .byte  1           ; All the booleans we need to keep track of
appleEaten:
    .byte  1
btnPress:
    .byte  1
gameOver:
    .byte  1
appleX:                   ; Coordinates of the apple
    .byte  1
appleY:
    .byte  1
mvIndex:
    .byte  1
snkLength:
    .byte  1
directions:
    .byte  64          ; Reserves the maximum possible number of directions
.cseg

reset:
    ldi    util,  (PLtch | PData | PClk | Gltch | GData | GC1k)
    out   DDRB,  util           ; Sets all of the SR pins to output
    call  initVar
    call  initArray

    cli           ; Disable global interrupt system
    call  T1Setup        ; Sets up the hardware and timer ints.
    call  INT0Setup
    sei           ; Re-enable global interrupt

    call  spawnApple      ; Spawn the first apple in the game
```

```

loop:
    ldi    ZH,    high(btnPress)           ; if INT isn't triggered, don't poll
    ldi    ZL,    low(btnPress)
    ld     data,   Z
    tst    data
    breq   skipChangeDir
        call   changeDir
skipChangeDir:

    ldi    ZH,    high(snakeMove)          ; Check if the timer has triggered
    ldi    ZL,    low(snakeMove)
    ld     data,   Z
    tst    data
    breq   skipUpdate
    ldi    ZH,    high(gameOver)          ; if the game is over, don't update
    ldi    ZL,    low(gameOver)
    ld     data,   Z
    tst    data
    brne   skipUpdate
        ldi    util,   FALSE
        ldi    ZH,    high(snakeMove)
        ldi    ZL,    low(snakeMove)
        st     Z,    util
        call   moveHead
                                         ; Move the head in the correct dir

        ldi    ZH,    high(appleEaten)       ; If an apple is eaten, skip the tail
        ldi    ZL,    low(appleEaten)
        ld     data,   Z
        tst    data
        brne   skipTail
        ldi    ZH,    high(gameOver)          ; If the game is over don't move tail
        ldi    ZL,    low(gameOver)
        ld     data,   Z
        tst    data
        brne   skipTail
        call   moveTail
                                         ; Function that moves the tail

skipTail:
    call   updateLength
                                         ; Update the position variables
                                         ; (Movement index, snake length, etc)

skipUpdate:
    call   shiftArray
                                         ; Constantly displays LEDs array

    rjmp  loop
    rjmp  reset

;***** Snake Moving *****

moveHead:
    push  util
    push  util2
    push  data

    cpi   headDir,LEFT
    brne  f1
    cpi   headX, 0
    breq  f1
                                         ; If the snake is going left...
                                         ; and is not at the left boundary...

```

```

        dec    headX           ; ... move the snake left!
        rjmp   doneMove        ; Then jump to the end

f1:
        cpi    headDir,RIGHT  ; If the snake is going right...
        brne   f2              ; and is not at the right boundary...
        cpi    headX, 7        ; ... move the snake right!
        breq   f2
        inc    headX
        rjmp   doneMove

f2:
        cpi    headDir,UP      ; If the snake is going up...
        brne   f3              ; ... and is not at the top boundary...
        cpi    headY, 0        ; ... move the snake up!
        breq   f3
        dec    headY
        rjmp   doneMove

f3:
        cpi    headDir,DOWN    ; If the snake is going down...
        brne   doneMove        ; and is not at the bottom boundary...
        cpi    headY, 7        ; ... move the snake down!
        breq   doneMove
        inc    headY

doneMove:
        ldi    ZH,    high(appleX) ; Grab the coordinates of the apple
        ldi    ZL,    low(appleX)
        ld     util,   Z
        ldi    ZH,    high(appleY)
        ldi    ZL,    low(appleY)
        ld     util2,  Z
        cp    headX, util      ; Compare the head with the apple
        cpc   headY, util2
        brne  skipEatApple
        ldi    util,   TRUE      ; Set appleEaten boolean to TRUE
        ldi    ZH,    high(appleEaten)
        ldi    ZL,    low(appleEaten)
        ld     util,   Z

skipEatApple:
        ldi    ZH,    high(LEDs) ; Look snake's next pixel
        ldi    ZL,    low(LEDs)
        add   ZL,    headY
        ld    util,   Z
        ldi    util2, 0x80        ; Grab row that the snake is going into
        mov    data,  headX      ; Left move pixel possible

shiftMask:
        tst    data
        breq  doneShifting     ; util2 = 0x80 >> headX
        lsr    util2
        dec    data
        rjmp   shiftMask

doneShifting:
        and   util,  util2
        breq  setLEDs          ; Skip the lose function
        call   lose

```

```

setLEDs:
    ldi    ZH,    high(LEDs)           ; Look at snake's new position
    ldi    ZL,    low(LEDs)
    add   ZL,    headY
    ld     util,   Z
    or    util2, util
    ldi   ZH,    high(LEDs)
    ldi   ZL,    low(LEDs)
    add   ZL,    headY
    st    Z,    util2
    pop   data
    pop   util2
    pop   util
ret

moveTail:
    push  data
    push  util

    ldi    ZH,    high(mvIndex)
    ldi    ZL,    low(mvIndex)
    ld     data,   Z
    ldi   ZH,    high(directions)
    ldi   ZL,    low(directions)
    add   ZL,    data
    ld    util,   Z
    cpi   util,   LEFT
    brne b1
    dec   tailX
    rjmp  doneMvTail
b1:
    cpi   util,   RIGHT
    brne b2
    inc   tailX
    rjmp  doneMvTail
b2:
    cpi   util,   UP
    brne b3
    dec   tailY
    rjmp  doneMvTail
b3:
    cpi   util,   DOWN
    brne doneMvTail
    inc   tailY

doneMvTail:
    ldi    ZH,    high(directions)    ; Store headDir in directions[mvIndex]
    ldi    ZL,    low(directions)     ;      "
    add   ZL,    data               ;      "
    st    Z,    headDir            ;      "

    ldi   util2, 0x80
    mov   data,  tailX
shiftTailMask:

```

```

tst    data          ; Count how many times we need to shift
breq   clrLEDs      ; Jump if we are done shifting
lsl    util2         ; Move over the correct # of pixels
dec    data          ;
rjmp   shiftTailMask ; Do it again

clrLEDs:
ldi    ZH,    high(LEDs) ; Load the current LEDs status in data
ldi    ZL,    low(LEDs)  ;
add   ZL,    tailY     ; Go to the row at headY
ld    data,  Z         ;

com   util2         ; We are going to zero the bit
and   data,  util2   ;
st    Z,    data      ;

pop   util           ;
pop   data           ;

ret

;***** Game Functions

changeDir:
push  util
push  data
push  looper
push  util2

ldi   util, FALSE    ; Turn the button press boolean off
ldi   ZH,  high(btnPress) ;
ldi   ZL,  high(btnPress) ;
st   Z,   util       ;

ldi   looper,0x08    ; Create a loop to read all 4 buttons

readBtn:
in    util, PINC    ; Read in the current state of the btns
and   util, looper  ; See if one of the buttons is pressed
brne  doneRead      ; If true, util holds the correct btn
lsl    looper        ;
brne  readBtn       ; Done reading

doneRead:
mov   data, util    ; We are manipulating the new data
add   data, headDir ; Add the previous & new direction

ldi   util2, (UP+DOWN) ; If new + old direction = UP+DOWN...
cpse  data, util2   ; ... the snake is trying to backtrack
rjmp  checkLR       ; Not backtracking? change direction
rjmp  dontSetDir    ;

checkLR:
ldi   util2, (LEFT+RIGHT) ;      " " "
cpse  data, util2       ;      " " "
mov   headDir,util     ; If not, set headDir to the new dir

dontSetDir:
pop   util2

```

```

pop    looper
pop    data
pop    util
ret

updateLength:
    ldi    ZH,    high(appleEaten)           ; Check if the apple is eaten
    ldi    ZL,    low(appleEaten)
    st     Z,    util
    tst    util
    breq   skipIncrLength
    ldi    ZH,    high(snkLength)
    ldi    ZL,    low(snkLength)
    ld     util,  Z
    inc    util
    st     Z,    util

    ldi    ZH,    high(directions)
    ldi    ZL,    low(directions)
    add    ZL,    util

    dec    ZL
    ldi    util,  high(directions)
shiftData:
    ld     data, -Z
    inc    ZL
    st     Z,    data
    dec    ZL
    cpi    ZL,    low(directions)
    cpc    ZH,    util
    brne   shiftData
                ; Compare the upper nibble
                ; If we've reached the bottom, we done.

    call   spawnApple
skipIncrLength:

    ldi    ZH,    high(mvIndex)             ; Retrieve the movement index
    ldi    ZL,    low(mvIndex)
    ld     util,  Z

    ldi    ZH,    high(directions)
    ldi    ZL,    low(directions)
    add    ZL,    vutil
    st     Z,    headDir
                ; look at the directions array
                ; Go to directions[mvIndex]
                ; Store the most recent headDir in the
                ; space (or over write the used move)

    inc    util
                ; Increment register holding mvIndex

    ldi    ZH,    high(snkLength)
    ldi    ZL,    low(snkLength)
    ld     data,  Z
                ; Retrieve the snkLength once again

    cp     util,  data
    brne   skipClrMV
    clr    util
                ; Clear mvIndex if = snkLength
skipClrMV:
    ldi    ZH,    high(mvIndex)
                ; Store the mvIndex back into RAM

```

```

ldi    ZL,    low(mvIndex)          ;  "
st     Z,     util                ;  "
ret

spawnApple:
    push  util
    push  looper
    push  data
    push  util2

    ldi   util, FALSE              ; Set the appleEaten variable to false
    ldi   ZH,   high(appleEaten)    ;  "
    ldi   ZL,   low(appleEaten)     ;  "
    st    Z,    util               ;  "

    ldi   looper,2                ; We are doing two conversions
doubleConv:
    ldi   util, 1 << ADEN | 1 << ADSC | 1 << ADPS1 | 1 << ADPS0
    sts  ADCSRA,util             ; Start the conversion, set prescaler
    ldi   util, 1 << ADC0D       ; Turn off the digital use of PC0
    sts  DIDR0, util              ;  "

waitConversion:
    lds   util, ADCSRA            ; See if the conversion is complete
    andi util, 1 << ADIF          ; Only look at the ADC Interrupt Flag
    breq waitConversion           ; If it isn't set, continue to poll

    lds   data, ADCSRA            ; Set the Flag bit to clear it (wack)
    sbr  data, 1 << ADIF          ;  "
    sts  ADCSRA,data              ;  "

    lds   data, ADCL              ; Reading in all the data
    lds   util, ADCH              ; Required, but not used
    andi data, 0x07                ; We only care about the 3 LSB

    dec   looper
    breq  setY
    ldi   ZH,   high(appleX)       ; We've done one conversion
    ldi   ZL,   low(appleX)         ; If it's already done
    st    Z,    data               ; Look at the X apple Coordinate
    rjmp doubleConv               ;  "
    ; Put the "random" number into X

setY:
    ldi   ZH,   high(appleY)       ; Look at the Y apple Coordinates
    ldi   ZL,   low(appleY)         ;  "
    st    Z,    data               ; Put the "random" number into Y

    ldi   ZH,   high(LEDs)         ; Got to LEDs[appleY]
    ldi   ZL,   low(LEDs)          ;  "
    add   ZL,   data               ;  "
    ld    util, Z                 ;  "

    ldi   ZH,   high(appleX)       ; Grab the X coordinate of the apple
    ldi   ZL,   low(appleX)         ;  "
    ld    looper,Z                ;  "
    ldi   util2, 0x80              ; Left move pixel possible

```

```

shiftMask2:
    tst    looper
    breq   doneShifting2           ; Shift until util2 = 0x80 >> appleX
    lsr    util2
    dec    looper
    rjmp   shiftMask2
doneShifting2:
    or     util,  util2          ; LED[appleY] | (0x80 >> appleX)
    ldi   ZH,    high(LEDs)       ; Turn on the LED to represent apple
    ldi   ZL,    low(LEDs)        ;
    add   ZL,    data            ; Go to LEDs[appleY] one last time
    st    Z,    util
    pop   util2
    pop   data
    pop   looper
    pop   util
ret

lose:
    push  util
    ldi   util,  TRUE             ; Set the gameOver variable
    ldi   ZH,    high(gameOver)   ;
    ldi   ZL,    low(gameOver)    ;
    st    Z,    util             ;
    ser   util                 ; Set the matrix completely on
    ldi   ZH,    high(LEDs)      ;
    ldi   ZL,    low(LEDs)       ;
setRow:
    st    Z+,    util            ; Increment the row and set the LEDs
    cpi  ZL,    low(LEDs+matrixWidth) ; If the row = top row, we're done
    brne setRow                  ;
    pop   util
ret

;***** Setup Functions *****

T1Setup:
    ldi   util,  T1ps256         ; Set game refresh rate
    sts   TCCR1B,util
    ldi   util,  1 << TOIE1      ; Enable Timer1 overflow interrupt
    sts   TIMSK1,util
ret

INT0Setup:
    ldi   util,  1 << ISC00 | 1 << ISC01 ; INT0 interrupt to detect RISING edge
    sts   EICRA,util
    ldi   util,  1 << INT0        ; Enable 1st Ext. interrupt
    out   EIMSK,util
    ldi   util,  1 << INTF0
    out   EIFR,util
ret

initArray:
    ldi   util,  DOWN            ; The snake starts downward

```

```

ldi ZH,    high(directions)      ;   "
ldi ZL,    low(directions)       ;   "
st Z+,    util                  ;   "
st Z+,    util                  ;   "
st Z,    util                  ;   "
clr util                         ; Clear the matrix
ldi ZH,    high(LEDs)
ldi ZL,    low(LEDs)
st Z,    util
ldi util, 3                      ; Set the initial snake length
ldi ZH,    high(snkLength)
ldi ZL,    low(snkLength)
st Z,    util
ret

initVar:
    ldi headDir,DOWN           ; Sets the initial direction of travel
    ldi headX, 3                ; Start the snake in the top left
    ldi headY, -1
    ldi tailX, 3
    ldi tailY, -4
ret

; **** SR Output

; Changes util, util2, data, addrs, & looper
shiftArray:
    push util
    push util2
    push addrs
    push data
    push looper

    ldi util2, matrixWidth      ; The start of a looping mask

pwrLoop:
    dec util2                   ; move onto the next address
    clr data                     ; Turn off all the power pins
    ldi addrs, PSR
    call shiftBits
    clr addrs                   ; Dummy register for the cpse
    ldi data, 0x80               ; Start creating the mask
    mov util, util2              ; We need to preserve

createMask:
    cpse util, addrs            ; If the counter = 0, don't shift mask
    lsr data                     ; Shift over the mask
    cpse util, addrs
    dec util
    brne createMask
    com data                     ; Invert the bits to gnd one pin
    ldi addrs, GSR
    call shiftBits               ; Talk to the Ground SR
                                ; Send out the ground data

    ldi ZH,    high(LEDs)        ; Start gathering the data for PSR
    ldi ZL,    low(LEDs)         ;   "
    add ZL,    util2             ; Find the right address in the array
    ld data, Z                   ; Read in the LED byte

```

```

ldi    addrs, PSR           ; Talk to the power SR
call   shiftBits          ; Send out the data
tst    util2               ; Address has reached zero? We're done
brne   pwrLoop

pop    looper
pop    data
pop    addrs
pop    util2
pop    util

ret

; Requires the "addrs" & the "data" registers to be set
shiftBits:
    ldi    looper,1          ; Setup the mask
    in    util, PORTB         ; Reads in the current state of port
    com   addrs
    and   util, addrs        ; We are going to clear
    com   addrs
    out   PORTB, util        ; Zero the latch bit
    lsr   addrs
    lsr   addrs              ; Get the bit back to normal
                            ; Output the zeroed-latch
                            ; Focus the bit on the clock pin

cycle:
    com   addrs
    and   util, addrs        ; Clear the clock bit
    com   addrs
    out   PORTB, util        ; Revert the data back
                            ; Output the data (Clock low)

    mov   util, data          ; We are going to be destroying data
    and   util, looper        ; And it with the mask to isolate a bit
    breq  data0
    in    util, PORTB         ; If the result is a zero, cbi data pin
    lsl   addrs
    or    util, addrs         ; Re-set util
    out   PORTB, util         ; Focus on the data pin now
                            ; Set the data bit high
                            ; Send the data out

    rjmp  setLoop            ; Jump to finish the loop stuff

data0:
    in    util, PORTB         ; Re-set util
    lsl   addrs
    com   addrs
    and   util, addrs        ; focus on the data bit
    com   addrs
    out   PORTB, util         ; Invert the data of pin to clr
                            ; Clear the data bit in util
                            ; Revert the byte so that it's normal
                            ; Send the sucker out

setLoop:
    lsr   addrs
    or    util, addrs
    out   PORTB, util         ; Focus on the clock
                            ; Set the clock high
                            ; Update the output

    lsl   looper
    brne cycle               ; Shift over to the next bit
                            ; If the mask has cycled, keep looping

    lsl   addrs
    lsl   addrs
    or    util, addrs         ; Get back over to the latch
                            ; Put the latch high

```

```

        out    PORTB, util
ret

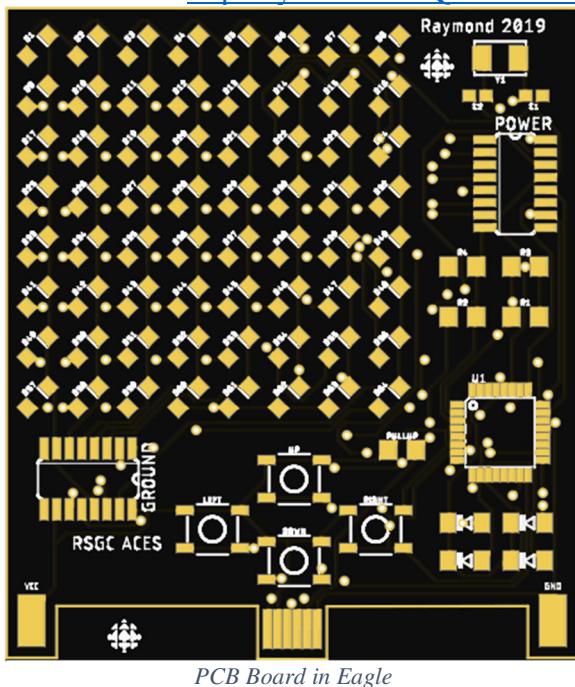
;***** Interrupt Vectors *****
INT0_vect:
    ldi    util,  TRUE
    ldi    ZH,    high(btnPress)
    ldi    ZL,    low(btnPress)
    st     Z,     util
ret

TIMER1_OVF_vect:
    ldi    util,  TRUE
    ldi    ZH,    high(snakeMove)
    ldi    ZL,    low(snakeMove)
    st     Z,     util
reti

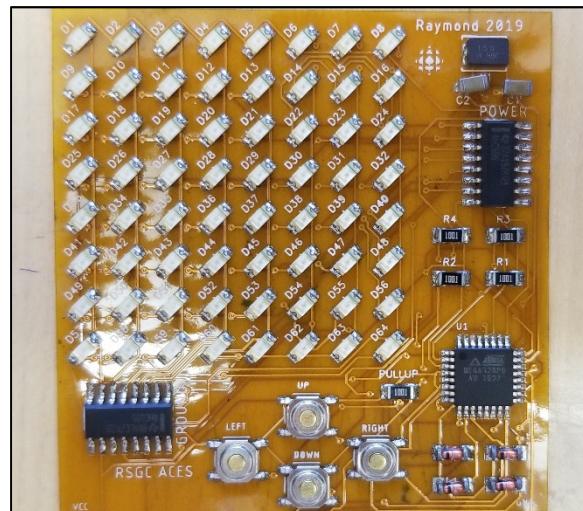
```

Media

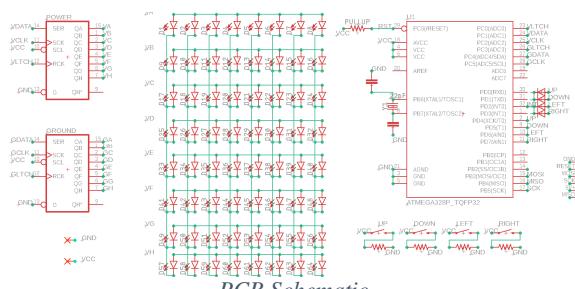
Personal Video: <https://youtu.be/lHOrBdLFWxc>



PCB Board in Eagle



Soldered PCB



PCB Schematic



Snake in action