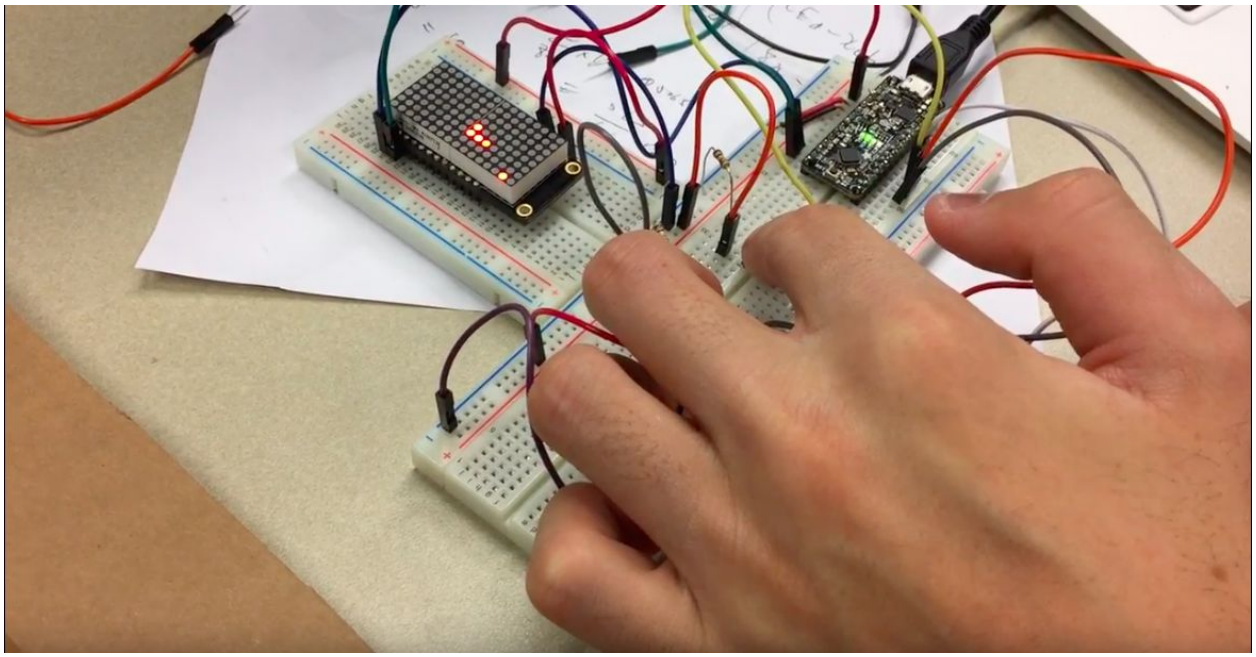
Mini Arcade Snake

Build your very own classic arcade game of Snake!

Do you miss classic games but all the arcades near you have all shut down? Are you a legend at Snake and want to show off your skills to your friends? Fear no more, after following this guide you can implement your own USB powered snake, thanks to help of embedded systems!



Mike Fu

CS 342

Professor Caitrin Eaton

December 4, 2018

Abstract

The goal of this project is to build the classic arcade game snake, controlled by the 2D analog joystick(or four buttons) and displayed on an LED matrix. The player moves a “snake” around the board, which is composed of some LEDs. The game will also spawn a random “fruit” on the board. If the snake “eats” the fruit, then it will grow in size by 1, and a new fruit will spawn. The game ends when the player’s snake head hits either the edge, or the body of the snake.

The input for this project is the 2D thumb analog, and the 4 buttons. I implemented both because some people might have one preference over the other. The 2D analog joystick has two inputs (one for the x-axis and one for the y-axis), and are connected to PC0, and PC1, respectively. The four buttons, which controls up, right, down, and left, are mapped to PD5, PD4, PD3, PD2, respectively.

The output for this project is the LED matrix. This is controlled by the I2C protocol, which is described later.

The inputs and outputs will be connected to GPIOs or General Purpose Input Output pins. These pins, on our microcontroller, can be used as both an input and as an output. We can use C++ to control how these pins behave. There are 3 main ports, B, C, and D, which controls how the 20 GPIO pins behave on the metro mini board. Each of these ports can be controlled writing to their respective PORT and DDR bits, in binary. Binary is basically how computers talk, and is equivalent to the base 2 system in math. DDR controls whether that bit (and its respective GPIO pin) will behave as an output or input, and PORT controls whether to output or not.

ADC, or analog to digital converter, is a way to read analog inputs. Analog inputs are essentially varying voltages. Essentially, what it does it map 2^n bins of values into n bits. The graph resembles the function y = floor(x), which is a series of steps. This also means that multiple voltages can be converted to the same number, as the function is not a one to one function. It is also important to note that most ADC sensors will not output the entire range of 0-1023, so it is very important to calibrate for maximum efficiency.

I2C, or two wire interface, is a way for a device to communicate with multiple devices at the same time. For my snake, the LED matrix uses I2C protocols to control all 128 LEDs with only two signal. I used a library, called LED backpack (approved by Caitrin) to communicate with the LED matrix. The I2C uses a data wire (PC4) and a clock wire (PC5).

The overall goals of this project was to create a working model of an arcade, which incorporates both a joystick and directional buttons. The buttons and joystick should be able to control the snake with minimal lag, and the game should follow the rules of snake. The snake game should also be very similar to the classic snake game that we grew up to love.

I am building this project because I really enjoyed playing snake as a kid, and I really want to take my embedded system skills to the next level by building something that is actually useful. My grandparents really like playing snake, so I am considering gifting them this project after I make it look pretty.

Features

All the following features can be viewed in the video section

- Displaying the score after the game ends
- Ability to switch between joystick and directional buttons in the middle of the game for preference
- Game speed increases as more and more fruits have been eaten

Parts

- ➢ Computer that has Arduino IDE
- ➢ 1 x Adafruit Metro Mini board
- ➢ 1 x 8 * 16 LED FeatherWing (0.8”)
- ➢ 2 x ultra-bright square 8*8 LED matrices (0.8”)
- ➢ 1 x Breadboard
- ➢ 1 x 2D Analog thumb Joystick
- ➢ 4 x Button Switch

- 4 x 10kΩ Resistors
- Wires

Additional Parts

These parts are optional, and will be used to create a mini game controller, which will be used in the future.

- Soldering iron
- Solder wires
- Cardboard
- Tape

Data Sheets & Resources

For this project, I will need the

- Metro mini schematic
- Atmega datasheet
- 2D analog joystick
- LED matrix backpack
- Adafruit Feather 32u4

In addition to following the Adafruit tutorial for setting up the LED matrix, I also followed a Tutorial for Snake, which helped me understand the underlying logic of snake a lot better.

The code and notes for this Friday's lab would also be handy.

Useful Figures

I will be referring to the following figures for more clarification on instruction:

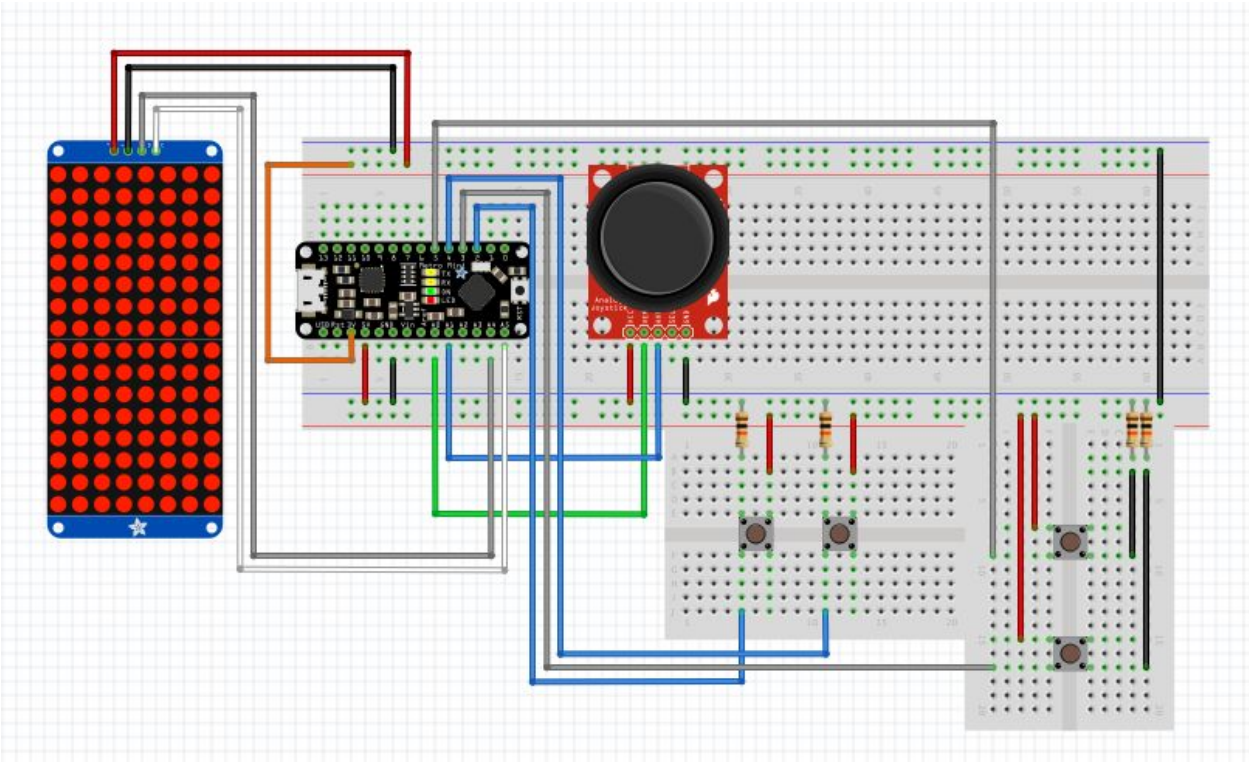


Figure 1: This is the basic hardware outline of arcade snake.

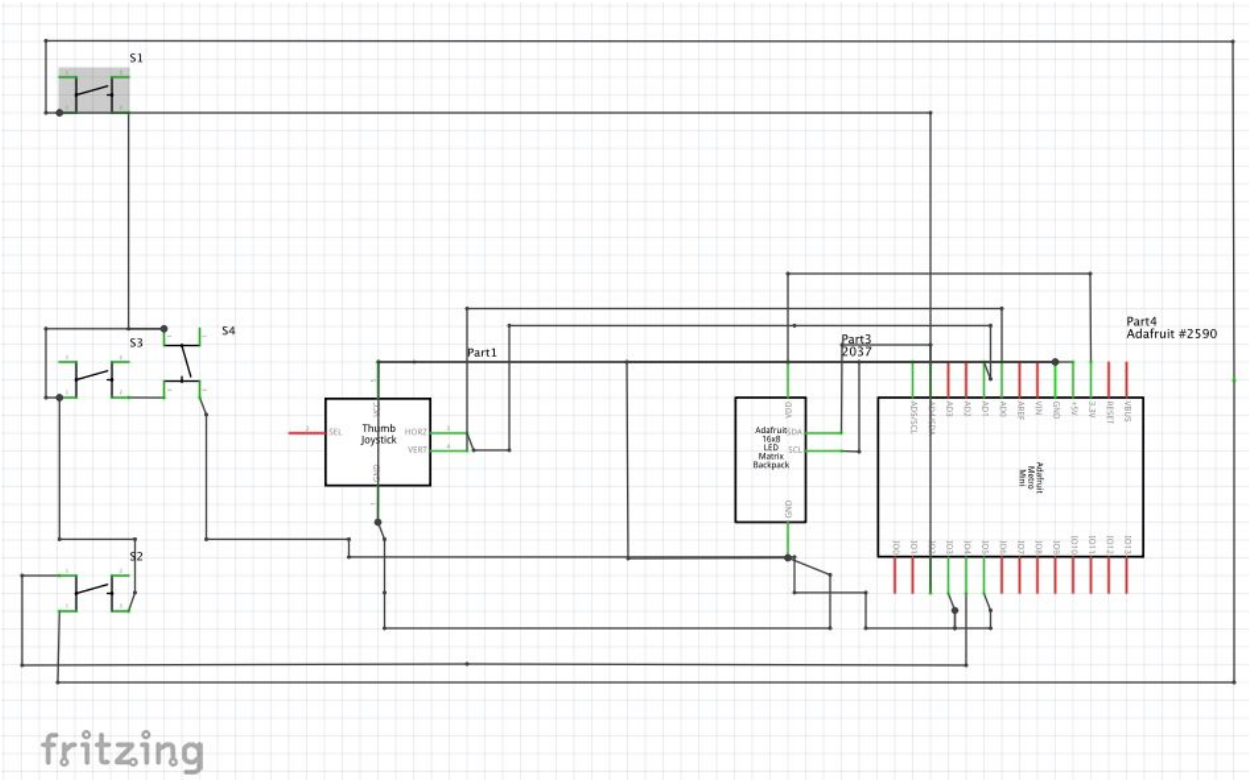


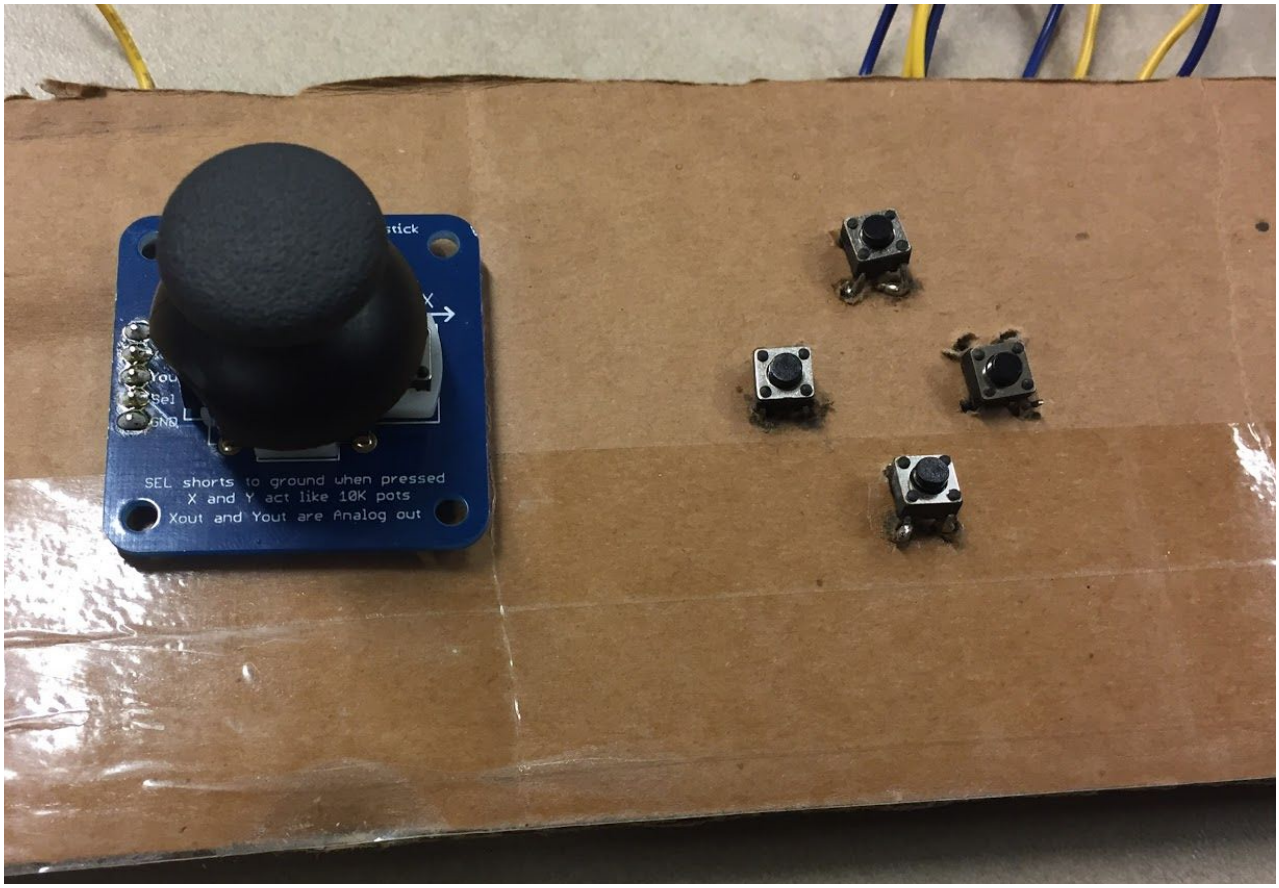
Figure 2: This figure shows the basic schematic of arcade snake

Procedure (Hardware)

The hardware of the alarm can be split into 4 main parts, which I will go over in detail.

1. The first part is setting up the LED matrix backpack. When I ordered my LED matrices with the FeatherWing, there was a bunch of soldering to do. I followed the "Overview" section of the link given in the datasheet section pretty closely. It is also important to view the Feather 32u4 pinouts to know which pins on the FeatherWing corresponds to which. Both datasheets can be found in the datasheets section. There is an awesome soldering guide available here from Adafruit that I followed religiously
2. Next, I configured LED matrices data wire to PC4, and clock wire to PC5. I also connected the 3V to power and ground to ground. *It is important to use the 3V output for the LED matrices.*
3. Next, I configured the 2D analog joystick. It has 3 input wires, but we are only going to use the X-direction wire and the Y-direction wire. Connect ground to ground, power to 5V and Xout to PC0 and Yout to PC1. More information of how to characterize the sensor can be found in the character sensoring section.
4. Next, I set up the buttons. I connected the up button to PD5, right to PD4, down to PD3, left to PD2. Be sure to also incorporate a pull down 10k ohms resistor, which will pull a small voltage down to 0. We need this because or else a small unwanted charge can cause a pin change interrupt.

Additional Hardware (Arcade keyboard)



To make the arcade keyboard, all you need are some solder and some cardboard

1. Map out where the pins from the analog joystick and the four buttons would go on the cardboard, and where they would come out on the other end
2. Poke holes in the cardboard so the wire would go through
3. Using a clipper to hold the wire and the joystick in place, solder a wire carefully onto each pin of the analog joystick. There is an awesome soldering guide available here from Adafruit that I followed religiously
4. Similarly, solder wires onto the two sides of the push button. I recommend soldering different colored wires to differentiate the different buttons
5. Visually verify that all solder looks good, and that none of the wires are touching each other

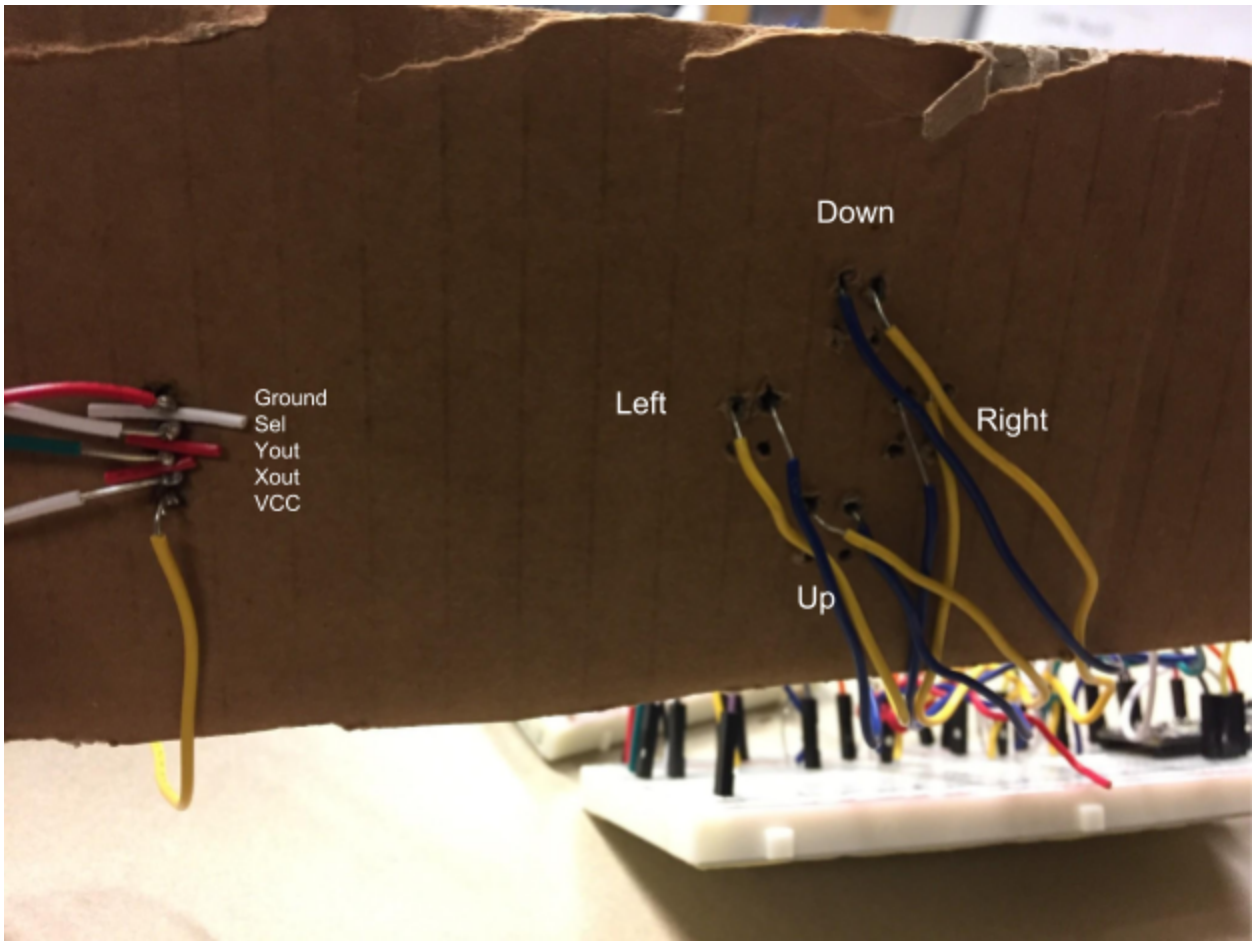


Figure 3: This figure shows the solder layouts for the four buttons and the analog joystick

Procedure (Software)

The software of the alarm can be split into 2 main groups, setting up the controls and configuring the logic behind snake, which I will go over in detail.

1. First, we set up the global variables that we are going to need for the game. This includes the width height, current location (both X and Y), the location of the "fruit" (both X and Y), and an array to hold the location of the tail (both X and Y). We also need two variables to hold the online mean for the X-axis sensor and the Y-axis sensor. I also created an enum for the four cardinal directions, and have a variable dir to hold the current direction.

2. We need to configure a timer for the overall control loop. I configured Timer0 so that it would reset every 1 ms. To do this, I set TCCR0A bit 1 to 1 to enable CTC mode. I also write 0x03 to TCCR0B to scale by 64, and set OCR0A to 250. I then enable TIMSK0 bit 1 to enable the ISR.

```
// enables TIMER0_COMPA
// 1ms per interrupt
TCCR0A = 0b00000010;    // CTC mode
TCCR0B = 0b00000011;    // scale by 64
OCR0A = 250;
TIMSK0 |= 0b00000010;
```

3. Next, we configure the pin change interrupts. We want each corresponding pin to correspond to the direction. The useButton boolean is to signify that the direction should be controlled by the directional buttons now.

```
ISR(PCINT2_vect) {

  // PD2
  if (PIND &= 0b00000100) {
    buttonDir = LEFT;
  }
  // PD3
  if (PIND &= 0b00001000) {
    buttonDir = DOWN;
  }
  // PD4
  if (PIND &= 0b00010000) {
    buttonDir = RIGHT;
  }
  // PD5
  if (PIND &= 0b00100000) {
    buttonDir = UP;
  }

  useButton = true;
}
```

4. Next, we set up the code to read from the ADC. In our ADC ISR, we want to calculate an online mean of the X-axis and the Y-axis. I used a window of 10 because the joystick is actually very accurate, but some outliers still occur. This will also minimize lag, which is important in a game.

```
ISR(ADC_vect) {
  int window = 10;   // small window because the joy stick is bloody accurate

  sensedADC = ADCL;                 // must read low byte first
  sensedADC |= (ADCH & 0x03) << 8;  // 10-bit precision

  if (senseX) {
    meanX = float(meanX * (window - 1) + sensedADC) / float(window);
  }
  else {
    meanY = float(meanY * (window - 1) + sensedADC) / float(window);
  }
}
```

We would also need to tell the ADC to switch from reading from the X-axis sensor to the Y-axis sensor every once in a while. I put this in my timer0 interrupt to flip a boolean that decides whether I will store the mean into meanX or meanY. I will also change the values of ADMUX so it would alternate readings from PC0 or PC1. SwitchTime dictates how of often I want to alternate readingsbetween the two analog pins. Since my Timer0 counts 1ms every tick, it would switch every 100ms.

```
ISR(TIMER0_COMPA_vect) {
  ms++;
  int switchTime = 100;

  if (senseX) {
    // sets ADC input to PC0
    ADMUX &= 0b11111000;
  }
  else {
    // sets ADC input to PC1
    ADMUX &= 0b11111000;
    ADMUX |= 0b00000001;
  }

  // switches between X and Y sensation every "switchTime" ms
  if (ms % (2 * switchTime) == 0) {
    senseX = true;
  }
  else if (ms % (2 * switchTime) == switchTime) {
    senseX = false;
  }

  // resets ms to avoid overflow
  if (ms == 10000) {
    ms = 0;
  }
}
```

Lastly, we configure the registers for ADC. I defaulted the reading from PC0 at first, enable the ADC interrupt, and set the prescaler to 16 (This is worked best for me from my experience). We also let ADC run on the Timer0A interrupt, which is every 1 ms.

```
/* Initializes ADC */
ADMUX = 0b01000000;   // read from PC0
ADCSRA = 0b11101100;  // enable ADC
ADCSRB = 0b00000011;  // on TIMER0A Interrupt
```

Now we need to configure the actual game of snake. The game can be broken down into 3 main parts, gameLogic, draw, and checkGameStatus

5. The gameLogic is the method that takes care of what should happen next based on a user input. Every “tick” of the game, it should update the head of the snake based on the direction, and make every tail of the snake follow the head. How I accomplished this was

setting the first element of the tail to the value of the current x and y location. Then I have iterate through the remain tail locations, updating the 2nd tail position to the 1st tail position, and updating the tail positions. This will essentially cause every element to be in the location of the element before it, which is what we want in snake.

Lastly, we update the head of the snake based on the direction.

```
void gameLogic() {

  int prevX = tailX[0];
  int prevY = tailY[0];
  int prev2X, prev2Y;
  tailX[0] = x;
  tailY[0] = y;

  for (int i = 1; i < nTail; i++) {
    prev2X = tailX[i];
    prev2Y = tailY[i];
    tailX[i] = prevX;
    tailY[i] = prevY;
    prevX = prev2X;
    prevY = prev2Y;
  }

  switch (dir) {
    case LEFT:
      y++;
      break;
    case DOWN:
      x++;
      break;
    case RIGHT:
      y--;
      break;
    case UP:
      x--;
      break;
  }
}
```

6. Next we draw the snake. This step is relatively straightforward: we draw the head, then iterate through the tailX and tailY arrays to draw the tail. We also want to draw the fruit. I am using the Adafruit_LEDBackpack.h library (approved by Caitrin) to draw pixels on LED matrix. All I would is call matrix.drawPixel(x,y) and specify the x and y locations.

```
void draw() {

  // draws snake
  matrix.clear();
  matrix.drawPixel(x, y, LED_ON);
  for (int i = 0; i < nTail; i++) {
    matrix.drawPixel(tailX[i], tailY[i], LED_ON);
  }

  // draws fruit
  matrix.drawPixel(fruitX, fruitY, LED_ON);
  matrix.writeDisplay();

}
```

7. Lastly, we need to check the game status. For example, if the snake head exits the boundary, or if the snake hits any of the tail, we need to set the gameOver boolean to true. We also need to extend the snake and increase the score if it eats a fruit.

```
void checkGameStatus() {

  // checks if the snake hit a bound
  if (x < 0 || x > width || y < 0 || y > height) {
    gameOver = true;
  }

  // checks if the snake has hit itself
  for (int i = 0; i < nTail; i++) {
    if (x == tailX[i] && y == tailY[i]) {
      gameOver = true;
      break;
    }
  }

  // checks if fruit has been eaten
  if (x == fruitX && y == fruitY) {
    score += 10;
    nTail++;
    fruitX = rand() % width;
    fruitY = rand() % height;
    delay_time = max(delay_time - 30, 100);
  }
}
```

Sensor Characterization

Because each sensor is innately different, you might get different ADC reading than me, which is why it is important to characterize your sensor.

Characterizing the 2D analog joystick is surprising easy. The joystick is actually composed of two ADC sensors, one for the X axis, and one for the Y axis. This means that we would have to characterize each axis separately, and configure admux so it would switch between evaluating both sensors.

To characterize an axis sensor, I arbitrarily chose 9 states: slightly left, most left, slight right, most right, slight up, most up, slight down, most down, and middle. At each state, I record the ADC reading, using our lab 7 code sensorCharacterization (which essentially calculates the ADC mean over 1000 tries). The results can be seen below:

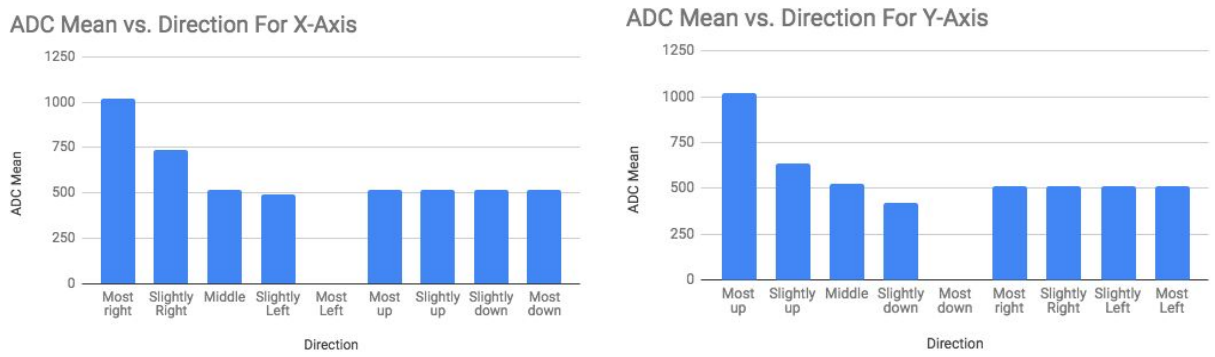


Figure 4: This figure graphically shows the ADC values for each location for each axis

X sensor						Y sensor				
Direction	ADC Mean	ADC Max	ADC Min	ADC Std		Direction	ADC Mean	ADC Max	ADC Min	ADC Std
Most right	1022	1023	1022	0		Most up	1022	1023	1022	0
Slightly Right	737	738	737	0		Slightly up	635	675	655	19
Middle	514	516	515	0		Middle	528	529	528	0
Slightly Left	489	494	490	1		Slightly down	419	420	419	0

Most Left	0	0	0	0		Most down	0	0	0	0
Most up	515	516	515	0		Most right	515	516	515	0
Slightly up	515	516	515	0		Slightly Right	515	516	515	0
Slightly down	515	516	515	0		Slightly Left	515	516	515	0
Most down	514	515	514	0		Most Left	514	515	514	0

Table 1: This table shows the sensed values and range for each location for each axis

Because these are arbitrary values, an equation wouldn't make much sense here.

As we can see, the 2D analog sensor is actually really accurate, because it has such a low standard deviation. This means that all recorded values are very close to each other. It also shows that if I move in a direction perpendicular to the axis that I'm measuring, the value measured will be the project on the said axis.

It appears that the sensor would be fair accurate if we map left to ADCx < 420 to left, ADCx > 750 to right, ADCy < 420 to down, and ADCy > 750 to up. Anything else would be considered in the middle.

For more information, check out the 2D analog sensor folder.

Results

One can test the game by playing it, and testing how well the sensors react (which is pretty well). For example, if you move the joystick right, it will always go right, and so forth. You can also test the game mechanics by running into the snake tail, or the sides, which would then terminate the game.

However, there are times when the system is unresponsive. Unfortunately, the LEDbackpack library does not work with timer interrupts, because the I2C protocol waits for the timer interrupt since to finish since the timer interrupt is of higher priority, and the timer interrupt can't finish without the I2C finishing. We have a case where two interrupts are waiting for each other to finish, in which case nothing gets done. Therefore, I have to use _delay_ms to stall the computer between ticks. I can definitely improve this project by writing my own I2C protocols, so I can use timer interrupts as well.

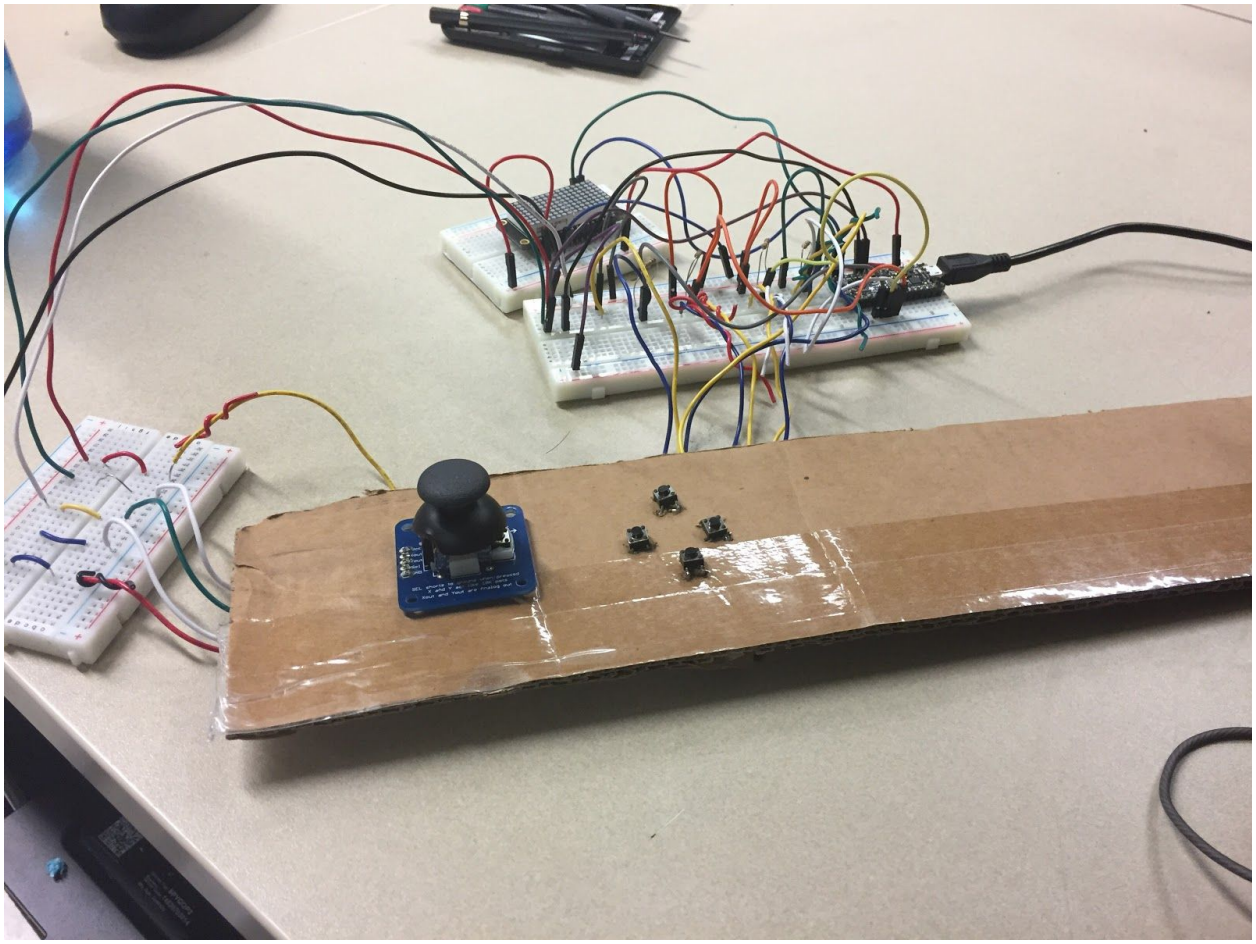
Discussion

Overall, I am very happy with the outcome of my project. I have successfully created an arcade snake game, playable with both a joystick and directional buttons. I have also made a mini game controller, which I can definitely use in the future.

Some things I could improve next time is to 3D print a controller board, so that it would look more aesthetically pleasing, and the wires won't get tangled as it does in my controller. Another improvement is to build an actual box to contain the "guts" of the project so no wires are being exposed. This will also take it one step further in to make it more like an arcade. I would also like to incorporate sound if I have time.

Lastly, I really wished I could have more time on this project, so I can incorporate other games, such as tetris, pong, and space shooters.

The real world application of this device would be a mini USB arcade game that you can carry around with you. It can be something to entertain while in the subway, or in class if you're ambitious enough.



Thank you for reading up to this far :) You made it!