Etch-A-Sketch Maze Generator

Final Report

Friday 13th, 2019
E155

Daniel Torres and Martha Gao

Abstract:

Mazes are fun to solve, but tedious to draw. This project aims to allow users to have the experience of solving a maze without having to construct it yourself. This device consists of an Etch-A-Sketch, stepper motors, an FPGA, and a microcontroller, all of which interface with each other to generate a predetermined maze on the Etch-A-Sketch screen, allowing the user to solve it manually at their own pace, preserving the physical interaction that makes mazes so fun, while maintaining the temporary nature of a digital device. The instructions to create the maze are stored in the microcontroller, which is also programmed with instruction decoders that are able to break down instructions into commands that identify which motor to move, in which direction, and for how long. These instructions are then sent to the FPGA, which then sends the correct command to the correct motor. The motors are attached directly to the Etch-A-Sketch knobs, and so when the motors turn, the knobs turn as well at a 1:1 ratio. Through this system, the Etch-A-Sketch can be controlled to produce a programmed drawing based on motor movements.

Introduction:

In an increasingly digital world, there is a certain nostalgia about playing with analog devices. The Etch-A-Sketch was an almost universal toy, known for its simplicity and precision, and has endured through the decades to be just as recognizable now as it was when it was first released back in the 1960s.

Puzzle games are popular with people of all ages. A maze is one of the most classic examples of a puzzle game, and it has been implemented in many mediums (gardens, paper, etc.). The point of a maze is to solve it, but it has to be somehow constructed in the first place. This project aims to remove that barrier to engaging in this puzzle, by allowing users to navigate their way through a pre-generated maze on the Etch-A-Sketch screen.

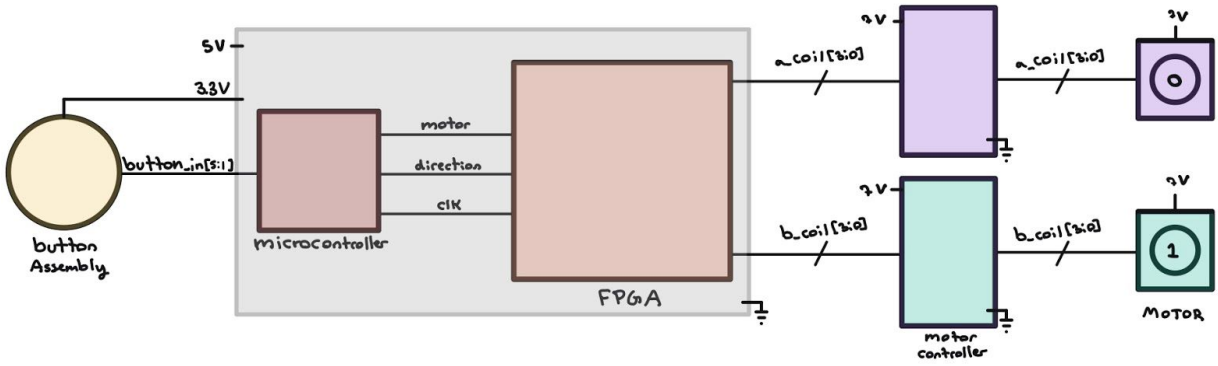


Figure 1. Block diagram of overall system

This project utilizes the precision that an Etch-A-Sketch provides, and interfaces the control knobs with a pair of stepper motors that are able to make very exact rotational movements. The degree and speed of rotation of these motors is dictated by instructions from the microcontroller, where they are stored.

The microcontroller is programmed with the maze instructions and instruction decoders. The instructions are translated from maze instructions to motor instructions and then sent to the FPGA, which is used to control both motors in both directions, one step at a time. Three mazes of varying difficulty are programmed into the device, and the desired maze is selected by a button with a corresponding LED on the breadboard: easy (green), medium (yellow), and hard (red).

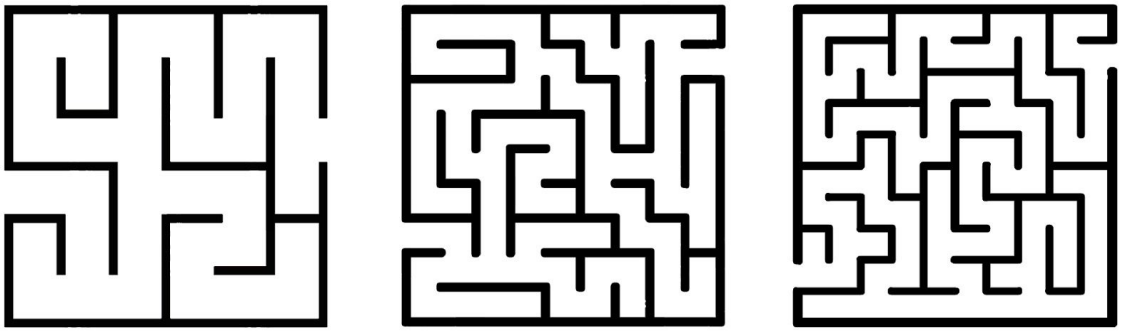


Figure 2. Selected mazes -- easy (left), medium (center), hard (right)

New Hardware:

A. Stepper Motors

Stepper motors were chosen for this project, due to their ability to perform precise movements in the angular position. A stepper motor is essentially a brushless DC motor that divides a full rotation into *n* equal steps. This way, the motor shaft can be programmed to only turn a certain number of steps, rather than the whole rotation. The motor does this in response to a train of input pulses.

A stepper motor is controlled by multiple toothed electromagnets surrounding a central piece of iron. The piece of iron is gear shaped in that it has teeth that align with the teeth of the electromagnets. When one electromagnet is given power, it pulls the teeth of the iron center into alignment with its own teeth. This causes the central teeth to me offset from the teeth of the next electromagnet. Thus, as the electromagnets receive power sequentially, the central piece is rotated in incremental steps. The electromagnets are grouped into complementary groups called phases, and each phase of electromagnets are powered together. The direction of rotation of the motor is controlled by the order in which these phases receive power.

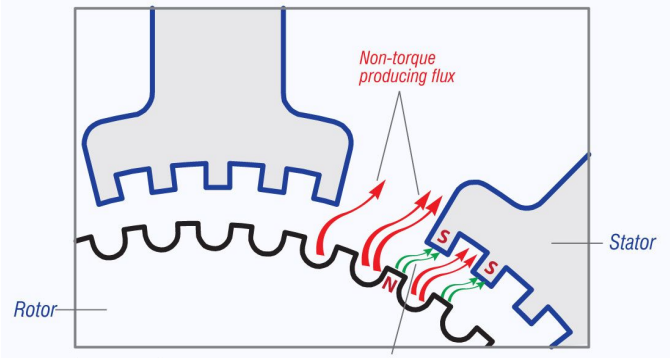


Figure 3. Stepper motor internal function

The most common motors are two phase stepper motors. The two basic winding arrangements of two-phase stepper motors are: uni-polar and bi-polar. Uni-polar stepper motors are generally easier to implement, as they have a center tap common wire. Using the common wire, the magnetic field (and therefore the direction of rotation) can be reversed easily without having to switch the direction of the current, and thus the driving circuit is made very simple.

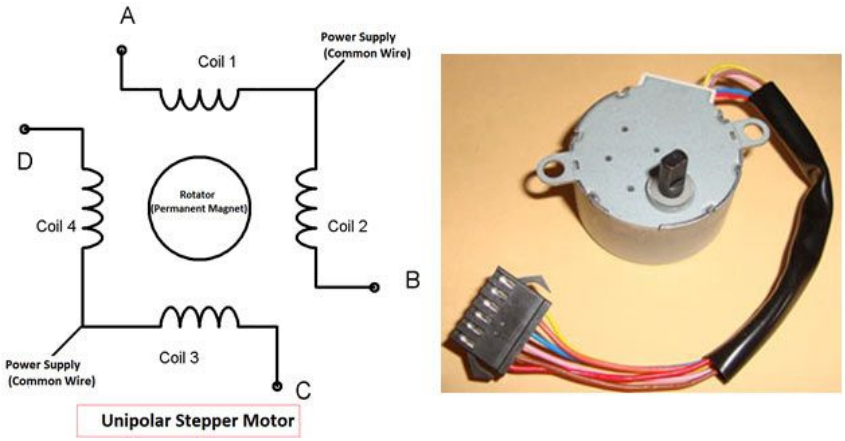


Figure 4. Uni-polar stepper motor

This project utilizes the Elegoo 28BYJ48 uni-polar stepper motor, which is one of the most popular hobbyist motors due to its ease of implementation and sufficient torque for most small projects. The package that was purchased also came with a ULN2003 driver board, which contains 7 NPN transistors which control the amount of current into the motor. These stepper motors have a step angle of 5.625º, and an internal gearbox reduction ratio of 1/64. The rated voltage is 5V, but responses from the manufacturer state the accepted voltage as up to 12V.

What was discovered through testing is that when run at 5V, the motors sometimes are not able to generate enough torque to actually start moving the knobs of the Etch-A-Sketch. This is because not enough current is flowing through the motors, and so to remedy this, the voltage was increased slightly to 7V to provide more current. This proved effective, as the motors do not stall anymore. Initially, an input voltage of 12V was attempted, but that resulted in the side effect of the motors themselves heated up much too quickly, and did not appear safe to run for long periods of time. 7V was found to be the lowest we could set the voltage, and still have the motors move the dials consistently for long periods of time.

For this project, two stepper motors and two ULN2003 driver boards are being used interfacing with the FPGA.

B. Etch-A-Sketch

The nostalgically fond Etch-A-Sketch was used in this project. Attached to the front of this toy was a screen and two knobs. The inside of the toy is filled with aluminum powder, which stays in place magnetically. Twisting the knob activates pulleys that move a stylus across the screen to displace the powder, allowing it to fall to the base of the toy. In doing so, the appearance of a dark line is left. By attaching the knobs to two stepper motors, the motors are able to control the movement of the stylus and and thus a programmed image is able to be drawn on the Etch-A-Sketch's screen. To erase the drawing, the whole assembly must be turned upside down and shaken, allowing the aluminum filament to fall back onto the screen, filling in the lines.

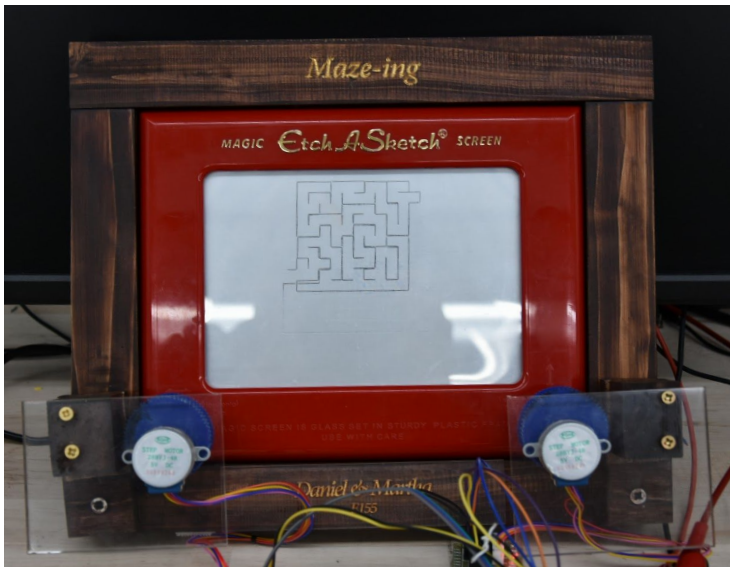


Figure 5. Etch-A-Sketch in final casing

Schematics:

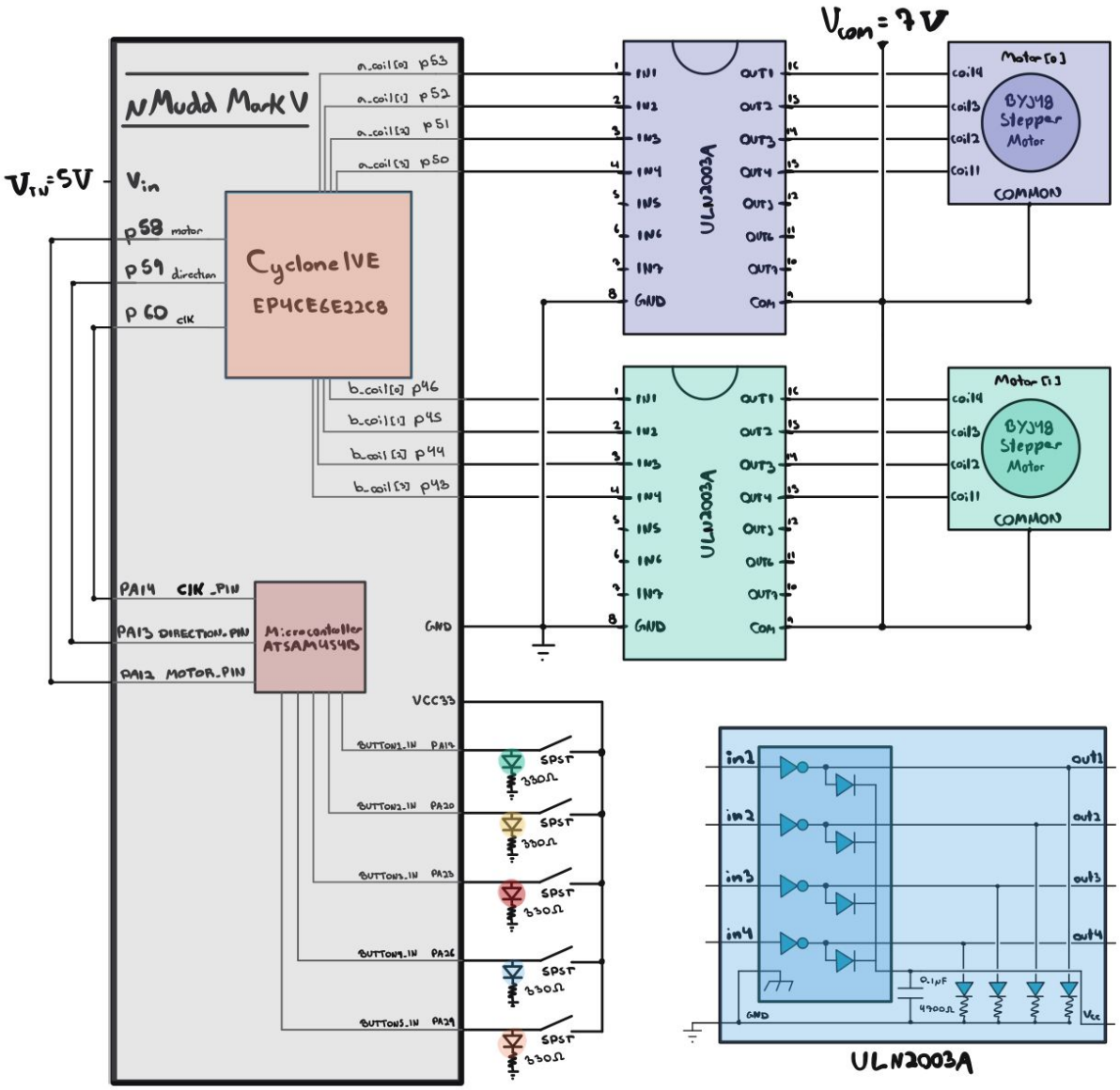


Figure 6. (Left) High level schematic of breadboard (Right) ULN2003 Driver Board circuit

Microcontroller Design:

The microcontroller is the main brain of the module. It has the mazes stored as arrays of instructions. It is then able to decode the instruction set and communicate these instructions to the FPGA. It has three main ways of drawing mazes. First the *drawStraightMaze()* function decodes the instructions and calls the *drawStraightLine()* function, which is then able to draw perfectly horizontal or vertical lines. The second maze type is a *drawRoundMaze()*, which calls *drawCircle()* which in tern calls *drawAngledLine()*. The third type of drawing function is the *drawPath()*, which takes in an array of coordinate points and draws a series of Bezier curves that draw a curve based on changing the weighted average between a series of four points.

The microcontroller begins with its *main()* function. This function simply initializes the clock, PIO, timer counter and pins. From here it enters an infinite loop while it waits for one of the button input pins to go high, indicating that a button has been pressed. Depending on the button being pressed, a different maze is drawn. Depending on the type of maze being drawn, a different maze-drawing function can be called. Three maze-drawing functions were written *drawStraightMaze()*, drawR*oundMaze()*, and *drawPath(),* with the instructions of each being encoded in a different way.

The *drawStraightMaze() function* takes in an array of strings, each containing two characters. This function runs on a while loop that runs for as long as the array length. During each loop, the string associated with the current step is decoded using three helper functions that return the duration of the step, which motor is being used, and if the motor will be turning clockwise or counterclockwise. This way the array of steps can remain legible with the code doing the brunt of the decoding. A length of 1 correlates to 650 (LINEAR_COEFF constant) calls to the *moveFPGA()* function, which correlates to about a 60º knob turn or a distance or about a centimeter. The penultimate step of this function calibrates the motor when the direction switches. Essentially, if the last direction moved is different from the current direction a distance of 350 (REVERSE_OFFSET constant) in the new direction is issued. This function call doesn't actually move the cursor, but moves the motor enough that the next *moveFPGA()* function call will move the cursor. This makes sure that the maze stays on top of itself when drawing the whole function. From here the *drawStraightLine()* function is called passing in the *motor*, *direction* and *distance* values.

The *drawStraightLine()* function uses a while loop to repeatedly input the current instruction into the *moveFPGA()* function. It calls this function a *duration* number of times. For reference, it takes about 1000 function calls for the motor to move 90º.

The *moveFPGA()* function digitally writes the *direction* and *motor* pins into the appropriate states. Then it puts the clk pin high. On the rising edge of this pin the FPGA's FSM changes states. Then we call an 800 microsecond delay to give the motor time to move. Finally the clk pin goes low.

Drawing the Straight Maze was all that was promised from our project proposal. The following functions are an extended feature set of the project, and were done for fun.

The *drawRoundMaze()* function takes in an array of strings, each containing 6 characters. Each string provides the instruction containing the type of instruction, and the parameters of the instruction. There are two types of instructions; both use a global variable *angle* that keeps track of the current angle of the cursor relative to the center point of the maze. 0º means straight left, 90º straight up and so on. For radially in or out movements, this value is used to determine which *lineAngle* to give the *drawAngledLine()* function. The length of this value is taken from the instruction set. Circular movements call the *drawCircle()* function, passing in the *radius*, *finalAngle*, and *instruction.* The *drawCircle()* function uses a while loop to draw a series of angled lines that together look like a circle to us. Each angled line is a degree change of 1. That is, that if the circle is being drawn clockwise, and the current angle is at 60º, it will use the radius determine the position at an angle of 61º. Then it determines the length and angle of the line needed to move into this position. It then passes this *lineAngle* and *distance* into the *drawAngledLine()* function. By continuously calculating this *drawAngledLine()* function

and updating the global angle, a circle can be drawn. The *drawCircle()* function halts when the *finalAngle* is equal to the global *angle.*

The *drawAngledLine*() function is the simplest function to explain, but also the longest function of the entire codebase. Given a *lineAngle* and *distance*, a line is drawn on the etch-a-sketch that matches the *lineAngle* and *distance.* Using some simple trig, It takes into account reverse direction calibration, cases where the line is fully horizontal or vertical, and ambiguous sines. The hardest part of this function is moving the two motors seemingly simultaneously to draw the appropriate angle. Say the function wants to move at an arbitrary angle. Using sin and cosine, we can calculate that for every x horizontal motor calls we want to make y vertical motor calls. From here a *globalAngleCount* variable is used that just counts upwards. Every time *globalAngleCount* reaches a multiple of y the motor is moved horizontally. Each time *globalAngleCount* reaches a multiple of x the motor is moved vertically. Notice how horizontal movements are based on the vertical component, and the vertical movements are based on the horizontal component. The variable *globalAngleCount* is global and is rarely reset, so that the count doesn't change as the desired angle changes. This property helps the drawn curves look smooth. In this way, this function is able to move the two motors at variable rates.

The *drawPath()* function uses a while loop to draw a series of Bezier Curves. Bezier Curves are essentially 4 coordinate points, where a curve is calculated based on a weighted average of the 4 points. The curve starts at point 0, and ends at point 3. When leaving point 0, it is traveling in the direction of point 1. When arriving at point 3, it is traveling from the direction of point 2. The function works by calculating the current and next x,y coordinate positions, and then calculating the angle and distance of a line that connects these two points. It then calls the *drawAngledLine*() function inputting this calculated *angle* and *distance.* Directions for the batman symbol are stored in the C code. Admittedly, due to time constraints this function was never properly implemented and is yet to be debugged. Had this function been properly finished the microcontroller would have been able to draw any path.

FPGA Design:

The FPGA has a simple but crucial role of keeping track of which position the stepper motor is in. If the stepper motor shaft and the FSM become desynchronized, then the motor will buzz but not spin. In order for the mazes being drawn to look good, the lines need to line back up with themselves. Thus it's critical that the lengths travelled by the etch-a-sketch's cursor remain constant. In this way, the task given to the FPGA is extremely important.

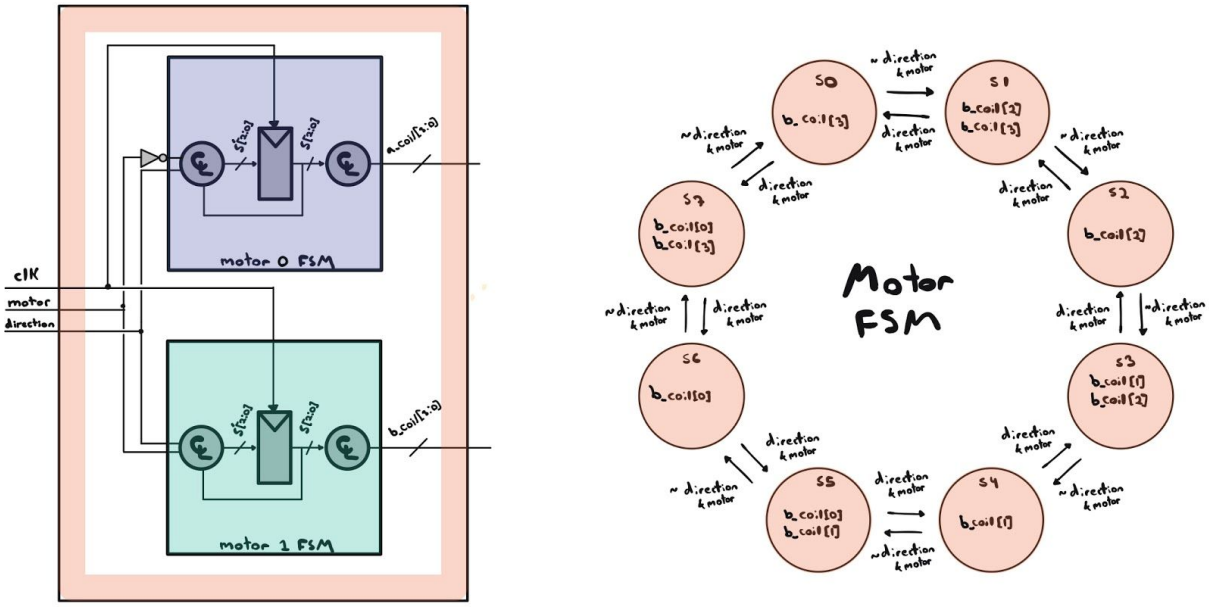


Figure 7. FPGA block diagram (left) FSM for motor logic (right)

The FPGA receives 3 bits of information from the microcontroller: *clk*, *motor,* and *direction*. The *motor* signal is 0 for the horizontal motor, and 1 for the vertical motor. Within the Motor FSM, it works like an enable. Thus the bit is inverted when inputted into the FSM module for motor 0. The *direction* bit indicates which direction the FSM should move. A direction of 0 indicates the the motor should spin clockwise, while a value of 1 indicates counterclockwise. On the rising edge of the clk, the direction and motor values determine in which direction the motor FSM should turn. The clk goes high each time the microcontroller's moveFPGA function is called. This allows the FSM to always be in sync with the microcontrollers commands.

Mechanical Design:

First, a housing for the Etch-A-Sketch was constructed, so that the Etch-A-Sketch would be immobile during the motor operations. It was designed to fit the Etch-A-Sketch very snugly, such that the Etch-A-Sketch was able to be inserted and removed with ease, but without any extra wiggle room to ensure the most accuracy for the drawing process. The structure was created out of wood machined according to measured specifications, and put together using screws.

Next, an assembly to attach the motors to the knobs was created. There were many design iterations for this component, as it is the most essential for accurately translating programmed code onto the Etch-A-Sketch screen. It is imperative to the quality of the final result that the motors are not able to shift during the drawing process by any degree of freedom, and

that the motors are perfectly vertically parallel to the knobs. The first design utilized a 1:1 gear set, one for each knob -- one part was press fit onto the Etch-A-Sketch knob, and the other was press fit onto the motor shaft. These parts were 3D printed, and were perfectly able to be securely press fit onto their respective parts without any slippage. Upon testing the gears, it became clear that this was not a viable option. When the teeth were pressed together such that no slip occurred, the torque needed to rotate the knob was too high for the motors and so they didn't turn, but when that pressure was slightly released, the gears were unable to catch each other securely and there was a lot of slipping. The next iteration utilized the fact that the gear structures were already very securely press fit onto their respective parts, and the gears were directly taped to each other such that the motors were now positioned directly above the Etch-A-Sketch knobs. This design produced a much better output, but was still not ideal because it relied on the motors being held immobile using our hands, which introduced human error, and that the tape was perfectly secure, which it was not. It also restricted the original intended design of allowing the user to manually solve the produced maze, since the motors were directly attached to the Etch-A-Sketch. The final design utilized the same idea of directly attaching the motors to the knobs as the tape, except the connection was made using magnets adhered to the respective gear structure of the knobs and the motors. The magnets chosen were strong enough to pull the center of the motors into alignment with the center of the knob, while also allowing them to be easily removed for manual play. The magnets were also strong enough that they did not slide against each other when one was rotated, which was essential as only one would be rotated by the motor, and the other (attached to the knob) would be rotated by the friction and magnetic attraction between them. One concern was that the magnets would interfere with the internal workings of the motor, as they rely on generated electromagnetic fields, but that was thankfully not the case.

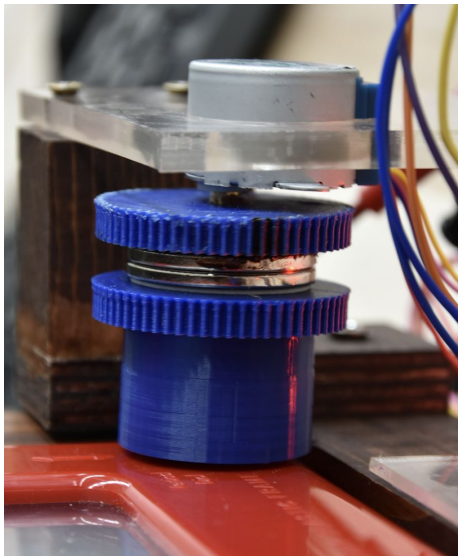


Figure 8. 3D printed knob and motor casing attached with magnets

Once the method of attaching the motors was determined, then came designing the method of attachment to the frame. This way, the system would be able to operate independently without using our hands to hold the motors still. The first idea was to have the motors attached to hinges that flipped the motors on and off the knobs, so that the motors could

be pulled off and snapped back on when needed. However, upon handling the magnets, this mechanism was deemed a safety hazard, as it could be very painful if the motors were swung onto a finger. As a result, a pivoting mechanism was constructed, that would have a structure attached to the frame on a pivot point, and the motors would be able to slide on and off the knobs. This design meant that some precision on centering the motors had to be sacrificed, and the motors would have to be adjusted once attached, but it was a necessary tradeoff to maintain safety.

Finally, the motor was attached to the pivoting structure using a sheet of acrylic that was laser cut to press fit around the motor, and then screwed onto the structure.

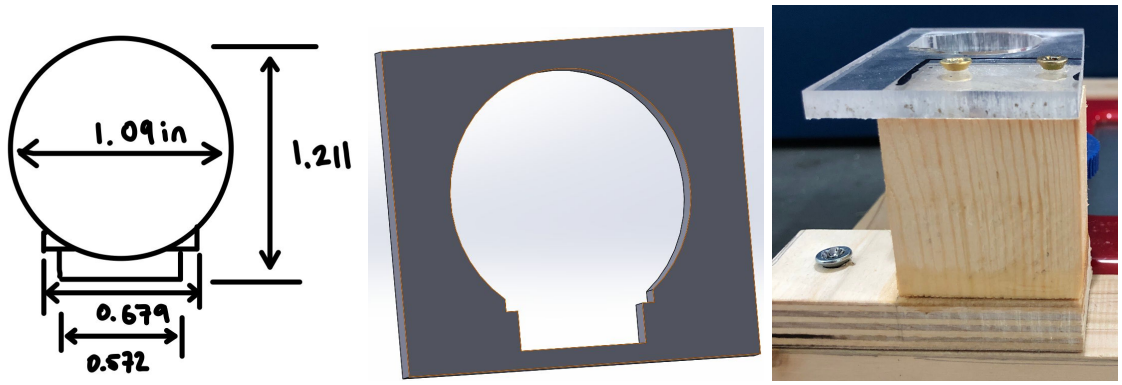


Figure 9.Motor dimensions sketch (left) CAD drawing for laser cut (center) Finished motor securing structure with pivot point (right)

Results:

We are very pleased with the outcome of our device. The system is quite robust, and successfully independently draws the three intended mazes reliably, and with a high level of precision. The system was able to both draw the initial line, as well as trace back over existing lines to return to a common spot on the maze almost perfectly, and certainly without interfering with the overall playability of the maze. User response was also very positive, in both the satisfaction gained from watching the Etch-A-Sketch stylus trace perfectly straight lines and return along the same exact path, and from the pleasure gained from being able to physically manipulate the knobs of the Etch-A-Sketch and complete the maze. At the core of this project, we aimed to create a system that brought joy to people and was fun to operate, so this was the most important metric of success for us.

Nevertheless, there is still much room for improvements if time and budget allowed. We had programmed in two stretch goal bonus mazes -- one was a circular maze, and the other was a much larger, much more complicated maze. The circular maze was successful once, when the motor casing was first attached to the frame but still held secure by our hands. However, we were not able to replicate this once the structure was firmly attached. We believe that this is due to the motors not being perfectly perpendicular, as well as the fact that the motors were becoming less reliable as they continued to be used, and were not able to produce the high level of precision needed to draw a circular shape. With more precise machining of parts, we believe that we could construct a secure structure for the motors that also ensure that

they are perfectly centered over the knob, and perfectly perpendicular to the knob, given more time.

The larger, more complicated maze required that the distance traveled for each space had to be scaled down, and was programmed with ~500 instructions. This maze always started off successfully, but only ever made it about a third of the way through the instructions before the motors started failing to rotate reliably. We believe that if we had more expensive and robust stepper motors, this maze could be completed.

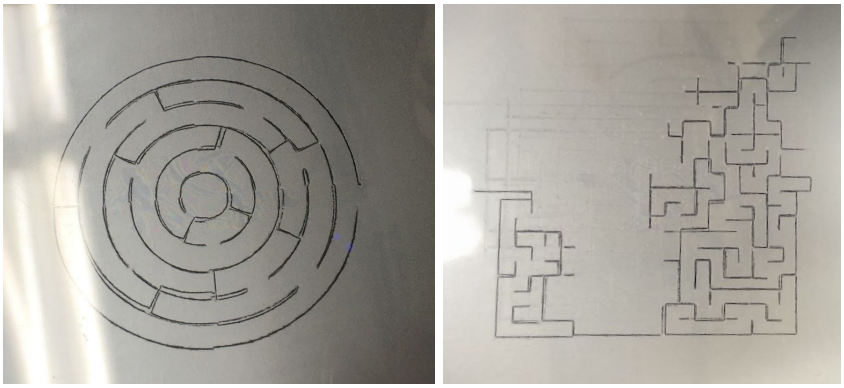


Figure 10. Circular maze (left) Partially completed scaled maze (right)

Overall, the system was very successful, and met all of our project specifications. Once we started attempting stretch goals, we saw all the improvements that could be made to the system that we would like to implement if given more time.

Figure 11. Completed structure!

References:

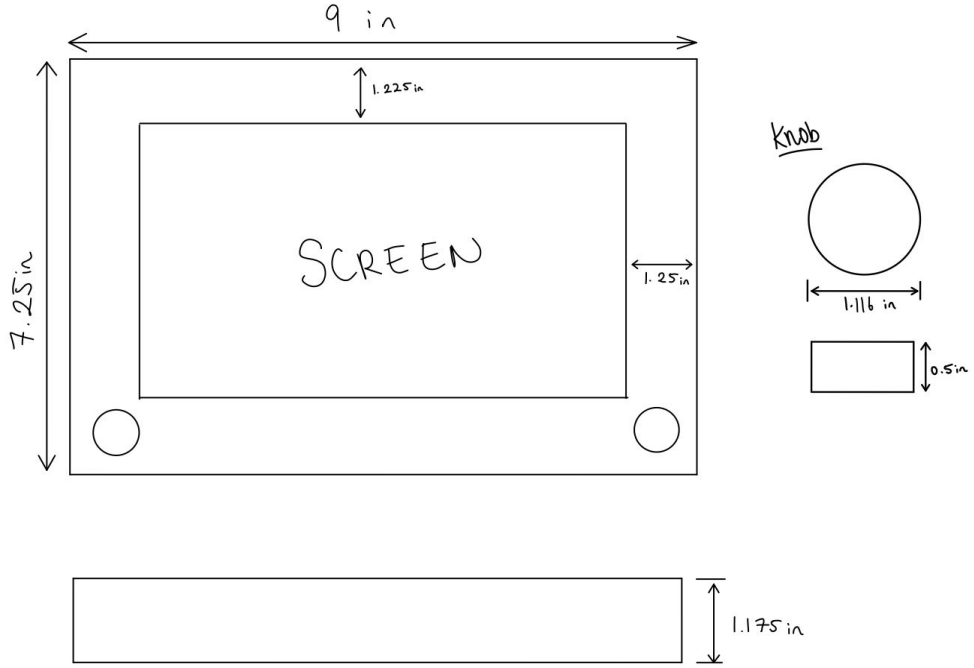
1. "ELEGOO 5 Sets 28BYJ-48 ULN2003 5V Stepper Motor + ULN2003 Driver Board for Arduino." *Amazon.*
2. "Stepper Motor." *Wikipedia*, Wikimedia Foundation, 13 Dec. 2019, en.wikipedia.org/wiki/Stepper_motor.
3. "Stepper Motor Interfacing with 8051 Microcontroller (AT89S52)." *CircuitDigest*, 26 July 2015, circuitdigest.com/microcontroller-projects/stepper-motor-interfacing-with-8051.
4. "Unipolar Stepper Motor vs Bipolar Stepper Motors." *Simply Smarter Circuitry Blog*, 31 Mar. 2016, www.circuitspecialists.com/blog/unipolar-stepper-motor-vs-bipolar-stepper-motors/.

Bill of Materials:

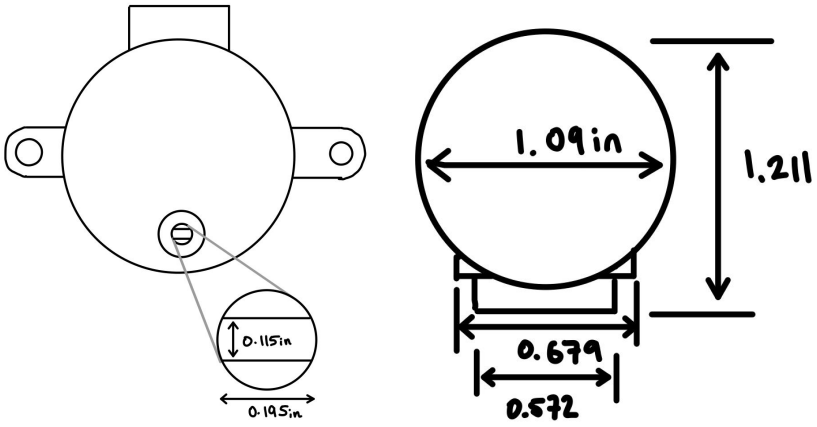
Part	Source	Vendor Part #	Price
Etch-A-Sketch	Amazon	20083951-6041762	$16.99
5x Stepper Motors & 5x Drivers	ELEGOO	28BYJ48 stepper motor ULN2003A driver	$13.99
6x 1.26" Disc Magnets with Double-Sided Adhesive	DIYMAG	HLMAG03	$8.99
8" Jumper wires	ELEGOO	EL-CP-004	$6.98
MicroPs board (μMudd and FPGA)	E155	N/A	$0.00
Wood beams, Plywood, Wood stain	Machine Shop	N/A	$0.00
Various screws	Engineering Stockroom	N/A	$0.00
⅜" Clear Acrylic	Machine Shop	N/A	$0.00
5x LEDs 5x 330Ω Resistors	Engineering Stockroom	N/A	$0.00

APPENDICES:

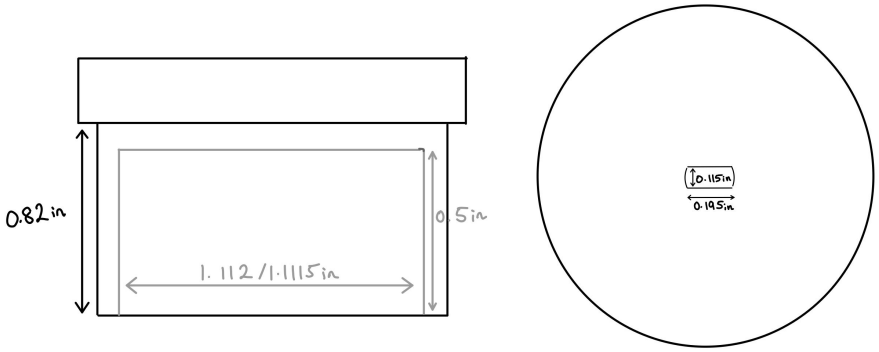
Etch-A-Sketch measurements:



Motor measurements:



Measurements for 3D printed knob and motor casings:



```
1   // code that draws maze on etch-e-sketch
2   // made by Martha Gao & Daniel Torres
3   // dtowersm@gmail.com
4   // mgao@g.hmc.edu
5   // Saturday November 16th 2019
6
7   #include "SAM4S4B.h"
8   #include <stdio.h>
9   #include <math.h>
10
11  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////
12  //-------------------------------------------------  Global Variables
    -------------------------------------------------//
13  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////
14
15  #define LOW  0 // Value to write a pin low (0 V)
16  #define HIGH 1 // Value to write a pin high (3.3 V)
17  #define PI 3.14159265
18
19  // PINS that communicate with the FPGA
20  #define MOTOR_PIN 12
21  #define DIRECTION_PIN 13
22  #define CLK_PIN 14
23
24  // Pins the communicate with the push buttons
25  #define BUTTON1_IN 17
26  #define BUTTON2_IN 20
27  #define BUTTON3_IN 23
28  #define BUTTON4_IN 26
29  #define BUTTON5_IN 29
30
31
32  // keeps track of how many more steps should be added for reversing direction
33  int INITIAL_OFFSET = 650;
34  // keeps track of how many steps should be present to start moving the motor
35  int REVERSE_OFFSET = 350;
36  // keeps track of how many steps should exist per unit of distance
37  int DISTANCE_COEFF = 650;
38
39  // keeps track of how many more steps should be added for reversing direction
40  int RADIAL_INITIAL_OFFSET = 650;
41  // keeps track of how many steps should be present to start moving the motor
42  int RADIAL_REVERSE_OFFSET = 250;
43  // keeps track of how many steps should exist per unit of distance
44  int RADIAL_DISTANCE_COEFF = 650;
45
46  // keeps track of the size of a degree step for making circles
47  // make this smaller to make circle more polished
48  double DEGREE_STEP = 1;
49
50  // global variable keeps track of the last direction the stepper motor traveled in
51  int last[2] = {2,2};
52
53  // global variable that keeps track of the angle for rotations
54  // goes from 0 to 359, with each change in one value being 5 degrees
55  // 0 is straight left
56  // 90 is straight up
57  // 180 is straight right
58  // 270 is straight down
59  double angle = 0;
60
61  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////
62  //-------------------------------------------------  Maze Encondings
    -------------------------------------------------//
63  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////
64
65  // tests the directions are correct up down left right
66  char testmaze[4][2] =
```

```
67      {"U1", "D1", "L1", "R1"};
68
69    // tests the distances of the lines, make sure 2 is twice as long as 1 etc
70    char testmaze2[15][2] =
71      {"D1", "R1", "L1", "D1", "R2", "L2", "D1", "R3", "L3", "D1", "R4", "L4","D1","R1", "L1"};
72
73    // tests changes in direction, to make sure the offset is correct
74    char testmaze3[10][2] =
75      { "L1", "R1", "L1", "R1", "L1", "R1", "L1", "R1", "L1", "R1"};
76
77    // tests changes in direction, to make sure the offset is correct
78    char testmaze4[11][2] =
79      { "L1", "U1", "L1", "U1", "L1", "R1", "D1", "R1", "D2", "U1", "R1"};
80
81    // saves directions of maze0
82    char maze0[47][2] = {
83    "R1",
84    "R1", "D1", "U1", "L1", "D2", "R3", "U2", "R1", "L1", "D2", "R3", "U2", "L1", "U1", "L2", "U2",
85    "D2", "R2", "U2", "D4", "L1", "R1", "U1", "R1", "U1",
86    "R1", "U1", "L1",
87    "U2", "L2", "D2", "U2", "L2", "D2", "L1", "U1", "D1", "R1", "U2", "L2", "D3", "R2", "D2", "U2", "L2",
88    "L1"};
89
90    // saves the directions of maze1
91    char maze1[104][2] =
92      { "R2", "L1", "D2", "R4", "U1", "L3", "R3", "D1", "R2", "U1", "D1", "R1",
93        "U2","L3", "R1", "D1", "U1", "R1", "U1", "L3", "U2", "R1", "L1", "D3", "U1", "R2",
94        "U1", "L1", "R1", "U2", "L3", "D1", "U1", "R2", "U1", "D1", "R1", "D3", "R1","D1","R1","D2",
95        "R2", "U2", "L1", "U1", "L1", "U1", "L1", "R1", "D1", "R1", "D2", "U1", "R1", "U5", "L1", "D3",
96        "U3", "R1",
97        "R1", "U1", "L1", // END
98        "L1", "R1", "U1", "L3", "D1", "U1", "L2", "D1", "R1", "D1", "R1", "D2", "R1", "U3", "D3", "L1",
      "U2",
99        "L1", "U1", "L1", "U1", "L4", "D2", "R3", "U1", "L2", "R2", "D1", "L3", "D4", "R2", "U1", "L1",
100       "U2", "D2", "R1", "D2", "U1", "L2",
101       "L1", //Begin
102     };
103
104   char maze2[129][2] = { "R1",
105   "R2", "U1", "R1", "U1", "L1", "U1", "L1", "R1", "D1", "R1", "D1", "L1", "D1", "L2", "D1", "R5", "R5",
106   "U5", "L2", "D1", "R1", "D3", "L1", "U2", "D2", "R1", "U3", "L3", "U1", "D1", "R1", "D2", "L1",
107   "D1", "R1", "L1", "U1", "L1", "R2", "U2", "R1", "U3", "L1", "U2", "D1", "L3", "D2", "U1", "L2",
108   "U1", "D1", "L1", "D1", "U1", "R3", "U2", "D1", "R3", "D1", "R1", "D2", "R2", "U3",
109   "R1", "U1", "L1",
110   "L1", "R1", "U1", "L2", "D2", "R1", "D2", "U2", "L1", "U2", "L2", "D1", "L1", "R1", "U1", "L3", "D2",
111   "U1", "L2", "D1", "U1", "R2", "U1", "L3", "D5", "R2", "U1", "R1", "D2", "R1", "U1", "R1", "U2",
112   "R1", "L1", "D1", "R2", "D1", "U1", "L2", "D3", "R1", "L1", "U2", "L1", "D4", "L1", "R2", "L1",
113   "U3", "L1", "U2", "L1", "D1", "L2", "D2", "R1", "D1", "U1", "L1", "D1",
114   "L1" };
115
116
117   // saves directions for very long maze
118   char mazeLong[495][2] = {"R3", "D1", "U1", "L2", "D5", "R1", "U1", "D1", "L1", "D4", "R5", "U1", "L4",
      "U1", "R1", "L1", "D1", "R2", "U3", "L1", "D1", "L1", "R1", "U1", "R3", "L1", "U1", "L1", "R1", "U1",
119   "R1", "L1", "U1", "L1", "D1", "U1", "L2", "D1", "R1", "D1", "U1", "L1", "U2", "D1", "R3", "D3", "L1",
      "D3", "R2", "D1", "R6", "U2", "R1", "U3", "L1", "R1", "U2", "L1", "R5",
120   "L2", "U2", "R1", "U2", "L1", "U3", "D2", "L1", "D4", "L1", "R1", "U2", "L3", "D3", "U1", "R1", "U1",
      "R1", "L1", "D1", "L1", "U3", "D1", "R4", "L1", "U2", "R1", "D1", "R1",
121   "D2", "L1", "D2", "L2", "D5", "R1", "D1", "L1", "R4", "L1", "U1", "R3", "D1", "R1", "L1", "U1", "L3",
      "D1", "L2", "U1", "L2", "D2", "R6", "U1", "D1", "R3", "U4", "L1", "D2", "U1", "L5", "D1",
122   "U3", "L1", "D2", "U2", "R1", "D2", "R5", "U1", "R1", "U4", "L1", "D1", "U1", "R1", "U2", "L1", "R1",
123   "R1", "U1", "L1",
124   "L1", "U2", "D2", "L1", "D5", "R1", "D1", "L1", "R1", "U1", "L2", "D2", "L2", "U2", "L2", "R3", "D1",
      "U1", "L1", "D2", "R3", "L1", "U5", "L1", "R1", "D1", "L2", "R2", "D2", "R1", "U3", "R1", "L1", "U1",
      "L2",
125   "U2", "R1", "D1", "U1", "R1", "U1", "D1", "L3", "U2", "R4", "L2", "D1", "L1", "R1", "U4", "D1", "R1",
      "U1", "R1", "L1", "U2", "L3", "R3", "D3", "L1", "D2", "L2", "U2", "R1", "D1", "U3", "R1", "L1", "D2" ,
126   "L1", "D1", "L2", "D1", "L4", "R1", "U1", "D2", "U1", "R3", "D1", "L2", "R1", "D2", "L1", "U1", "L2",
      "D3", "U2", "L1", "U1", "D1", "R1", "U1", "R2", "D1", "R1", "U2", "R1", "U3", "D1", "R2", "D3", "R1",
      "D2", "R2",
127   "U1", "R2",
128   "U5", "L1", "D1", "L1", "R1", "U1", "R1", "U4", "L1", "D2", "U2", "L5", "D1", "U1", "L2", "D1", "U1",
```

```
    "L4", "D1", "R3", "L1", "D2", "R1", "U1", "D1", "R1", "D2", "U2", "R2", "D1", "U1", "R1", "U1", "L3",
    "R1", "U1",
129 "D1", "R2", "D1", "L5", "U2", "L2", "U1", "L5", "D1", "U1", "L1", "D3", "L1", "U2", "D3", "U1", "R1",
    "U1", "R2", "U1", "R2", "D2", "R2", "D1", "U1", "R3", "U1", "D1", "R1", "U2", "L2", "D1",
130 "L2", "U2", "L2", "D5", "R2", "U1", "D1", "L2", "D2", "R2", "U1", "R1", "U1", "R3", "D1", "R1", "L1",
    "U1", "L1", "U1", "L2", "R1", "U1", "L1", "R1", "D1", "R1", "D1", "L2", "D1", "R2", "L1",
131 "D2", "R4", "U1", "R3", "L1", "U2", "R1", "U1", "R2", "D1", "U1", "L2", "U2", "L2", "U1", "D1", "R2",
    "D3", "L2", "U1", "L1", "R1", "D1", "R1",
132 "D2", "L3", "U1", "L1", "D1", "L1", "R1", "U1", "R1", "D1",
133 "R1", "D1", "L1", "D2", "R1", "L1", "D1", "L1", "R3", "L1", "D1", "R2", "U2", "L1", "U2", "D1", "L1",
    "R1", "D1", "R1", "D2", "R1", "D2", "L1", "D1", "U1", "R3", "L1", "D2", "R1", "D1", "U1", "L1", "U2",
    "L1",
134 "U2", "L3", "U1", "L1", "U3", "L3", "D2", "L1", "U2", "D2", "R3", "U1", "L1", "R1", "D1", "L1", "D2",
    "R2", "D1", "R3", "L1", "D1", "L1", "R1", "D2", "R2", "U1", "D3", "L2", "R1", "U1", "D1", "R1", "U2",
    "L2", "D1",
135 "L1", "D2", "R4", "U2", "D2", "L4", "U4", "L1", "R1", "D2", "L2", "D1", "R1", "L1", "U1", "R3", "U4",
    "L2", "U1", "L2", "U2", "L1", "U3", "L2", "D3", "L1"};
136
137 // CW for Clockwise
138 // CC for counterclockwise
139 // Radius
140 // Final Angle ()
141
142 // IN for traveling in
143 // OU for traveling out
144 // Distance to Travel
145
146
147 // draws concentric circles
148 // used for calibrating the in and out distance
149 char circletest1[17][6] =
150   { "CW6180", "CW6000",
151     "IN1000",
152     "CW5180", "CW5000",
153     "IN1000",
154     "CW4180", "CW4000",
155     "IN1000",
156     "CW3180", "CW3000",
157     "IN1000",
158     "CW2180", "CW2000",
159     "IN1000",
160     "CW1180", "CW1000"};
161
162 // one of the mazes of the code
163 char circlemaze[64][6] =
164   { "CC6180",
165     "IN1000",
166     "CW5210", "CC5050", "CW5120",
167     "IN1000",
168     "CW4140", "CC4070", "CW4090",
169     "IN1000",
170     "CW3200", "CC3030",
171     "OU1000",
172     "CC4350", "CW4060", "CC4030",
173     "IN1000",
174     "CC3330",
175     "OU1000",
176     "CC4310", "CW4340",
177     "OU1000",
178     "CW5040", "CC5250",
179     "IN100",
180     "CW4300", "CC4250",
181     "OU100",
182     "CC5220", "CW5340",
183     "IN1000",
184     "CC4330",
185     "IN1000",
186     "CC3290",
187     "IN1000",
188     "CC2090", "CW2120",
189     "IN1000",
190     "CC1030",
```

```
      "OU1000",
      "CW2080", "CC2310", "CW2030",
      "IN1000",
      "CC1140", "CW1120",
      "OU1000",
      "CW2290",
      "OU1000",
      "CC3210",
      "OU1000",
      "CC4140", "CW4240", "CC4210",
      "IN1000",
      "CW3090",
      "OU1000",
      "CW4120",
      "OU1000",
      "CW5180",
      "OU1000",
      "CC6010",};
// a series of coordinate points that generate bezier curves
// that draw the batman symbol
double batManSymbol[74] =
  { 0,0, 0,0.5, 1,1.75, 2,1.75,
    1, 1.25, 2, 0.5, 3, 0.75,
    3.13, 0.88, 3.25, 1.25, 3.25, 1.5,
    3.25, 1.25, 3.25, 1, 3.5, 1,
    3.75, 1, 3.75, 1.25, 3.75, 1.5,
    3.75, 1.25, 3.88, 0.88, 4, 0.75,
    5, 0.5, 6, 1.25, 5, 1.75,
    6, 1.75, 7, 0.5, 7, 0,
    6, 0.25, 5.25, 0.25, 5.25, -0.5,
    4.25, 0, 3.63, -1, 3.5, -1.25,
    3.37, -1, 2.75, 0, 1.75, -0.5,
    1.75, 0.25, 0.5, 0.25, 0, 0};

////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
//-----------------------------------------  Motor Moving Function
-----------------------------------------------//
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////

// initializes the Pins
int pinInit() {
  // initilialize pins that talk to FPGA
  pioPinMode(MOTOR_PIN, PIO_OUTPUT);
  pioPinMode(DIRECTION_PIN, PIO_OUTPUT);
  pioPinMode(CLK_PIN, PIO_OUTPUT);

  // initialize pins used to check for button presses
  pioPinMode(BUTTON1_IN, PIO_INPUT);
  pioPinMode(BUTTON2_IN, PIO_INPUT);
  pioPinMode(BUTTON3_IN, PIO_INPUT);
  pioPinMode(BUTTON4_IN, PIO_INPUT);
  pioPinMode(BUTTON5_IN, PIO_INPUT);
  return 1;
}

// this move function delivers the instructions
// to the FPGA. The FPGA's FSM's control the individual motors in this case
// motor tells the FPGA which motor to move
// direction tells the motor which direction to move in
// 0 means clockwise, 1 means counterclockwise
int moveFPGA (int motor, int direction) {
  // write the pins
  pioDigitalWrite(MOTOR_PIN, motor);
  pioDigitalWrite(DIRECTION_PIN, direction);
  // the FPGA's FSM changes states on the rising edge of the clk pin
  pioDigitalWrite(CLK_PIN, HIGH);
  // wait for the motor to move;
  tcDelayMicroseconds(800);
  pioDigitalWrite(CLK_PIN, LOW);
  return 1;
```

```
260   }
261
262
263   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
264   //-----------------------------------------  Line Drawing Functions
      -----------------------------------------------//
265   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
266
267   // helper function used to calibrate the distance of the verticle and horizontal components
268   // of the draw angled line function
269   // distance is the intended travel distance
270   double angledComponentDistanceCalibration(double distance) {
271     double output = RADIAL_INITIAL_OFFSET; // base value to make the motor start moving
272     output += (distance-1) * RADIAL_DISTANCE_COEFF; // add amount per step
273     return output;
274   }
275   // helper function that draws a line in the cardinal directions
276   int drawStraightLine(int motor, int direction, double duration) {
277     int d = 0;
278     while(d < duration) {
279       // move the motor
280       moveFPGA(motor,direction);
281       // update the last direction the motor traveled in
282       last[motor] = direction;
283       // gives time to let the motor move
284       ++d;
285     }
286     return 1;
287   }
288
289   // global Angle Count is just a counter, it is a global variable used
290   // to determine how often each motor should move relative to one another
291   // by making the variable global the count is saved between function calls
292   // which makes the function more consistent for smaller circles
293   unsigned int globalAngleCount = 0;
294
295   // helper function draws lines in non-cardinal directions
296   // used both to draw small steps of the circle and for the radial lines of the radius
297   // argueably the most important function of in this file
298   int drawAngledLine(double distance, double lineAngle) {
299
300     // keeps track of the the direction of each motor
301     // 0 is clockwise
302     // 1 is counter clockwise
303     int motorDirection[2] = {0,0};
304     if ( (lineAngle <= 90) || (lineAngle >= 270))
305       motorDirection[0] = 1;
306     if (lineAngle >= 180)
307       motorDirection[1] = 1;
308
309     // reverse direction calibration!
310     if (last[0] != motorDirection[0]) //horizontal
311       drawStraightLine(0, motorDirection[0], RADIAL_REVERSE_OFFSET);
312     if (last[1] != motorDirection[1]) // verticle
313       drawStraightLine(1, motorDirection[1], RADIAL_REVERSE_OFFSET);
314
315     // cases in which the line is horizontal or verticle
316     if (lineAngle == 90) {
317       drawStraightLine(1, 0, angledComponentDistanceCalibration(distance));
318     } else if (lineAngle == 180) {
319       drawStraightLine(0, 0, angledComponentDistanceCalibration(distance));
320     } else if (lineAngle == 270) {
321       drawStraightLine(1, 1, angledComponentDistanceCalibration(distance));
322     } else if (lineAngle == 0) {
323       drawStraightLine(0, 1, angledComponentDistanceCalibration(distance));
324     } else {
325       // modify triangleLineAngle to be between 0 and 90 to avoid ambiguous sine
326       double triangleLineAngle = lineAngle;
327       if (triangleLineAngle > 90 && triangleLineAngle <= 180)
328         triangleLineAngle = 180 - triangleLineAngle;
```

```
329       else if (triangleLineAngle > 180 && triangleLineAngle <= 270)
330          triangleLineAngle = triangleLineAngle - 180;
331       else if (triangleLineAngle > 270)
332          triangleLineAngle = 360 - triangleLineAngle;
333
334       // get horizontal and verticle components of the triangle
335       double horizontalSlope =   cos(PI * triangleLineAngle / 180);
336       double verticleSlope   =   sin(PI * triangleLineAngle / 180);
337
338       // determine the distance needed to be traveled in the horizontal and veritcal directions
339       // takes into account the last direction each motor turned in
340       int horizontalDistance = angledComponentDistanceCalibration(distance * horizontalSlope);
341       int verticleDistance   = angledComponentDistanceCalibration(distance * verticleSlope);
342
343       // used to determine how often in the global Angle count
344       // each motor should turn,
345       // horizontalCountRatio is based on verticleSlope
346       // verticleCountRation is based on hoizontalSlope
347       // they are based on the opposite component
348       int horizontalCountRatio = 1000 * verticleSlope;
349       int verticleCountRatio = 1000 * horizontalSlope;
350
351       // used to know how many steps the while loop should take,
352       // d goes up every time a motor is moved, once the total distance each motor needs to travelled
353       // is satisfied, the while loop will break
354       int d = 0;
355       int totalDistance = verticleDistance + horizontalDistance;
356
357       // while loop to move the motor
358       while(d <= totalDistance) {
359         // move horizontal motor
360         if (globalAngleCount%horizontalCountRatio == 0) {
361           moveFPGA(0, motorDirection[0]);
362           ++d;
363         }
364         // move verticle motor
365         if (globalAngleCount%verticleCountRatio == 0){
366           moveFPGA(1, motorDirection[1]);
367           ++d;
368         }
369         // global Angle Count is just a counter, it is a global variable used
370         // to determine how often each motor should move relative to one another
371         // by making the variable global the count is saved between function calls
372         // which makes the function more consistent for smaller circles
373         ++globalAngleCount;
374       }
375     }
376     // update last direction each motor moved
377     last[0] = motorDirection[0];
378     last[1] = motorDirection[1];
379     return 1; // returns one for debugging puposes
380   }
381
382   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
       ////////////////
383   //-----------------------------------------   Straight Maze Maker
       ------------------------------------------------//
384   ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
       ////////////////
385
386
387   // helper function that decodes the direction from
388   // the list of instructions
389   // 0 is clockwise
390   // 1 is counterclockwise
391   int decodeDirection(char in[2]) {
392     if (in[0] == 'L') {
393       return 1;
394     }
395     if (in[0] == 'D') {
396       return 1;
397     }
```

```
398     return 0;
399   }
400
401   // helper function that decodes which motor should
402   // be used based on the list of instructions
403   int decodeMotor(char in[2]) {
404     if (in[0] == 'U') {
405       return 1;
406     }
407     if (in[0] == 'D') {
408       return 1;
409     }
410     return 0;
411   }
412
413   // helper function that decodes the duration from
414   // the list of instructions
415   int decodeDuration(char in[2], int motor, int direction) {
416     int output = INITIAL_OFFSET; // base value to make the motor start moving
417     int x = in[1]-'0';  // the -'0' is to convert from char to int
418     --x; // so that the distance coefficient only effects duration>1
419     output += (x * DISTANCE_COEFF); // add amount per step
420     return output;
421   }
422
423   int drawStraightMaze( char maze[][2], int arraylength) {
424     int i = 0;
425     while(i < arraylength) {
426       // get values from list
427       int motor     = decodeMotor(maze[i]);
428       int direction = decodeDirection(maze[i]);
429       int duration  = decodeDuration(maze[i], motor, direction);
430
431       // goes in opposite direction if needed
432       if (last[motor] != direction) {
433           drawStraightLine(motor, direction, REVERSE_OFFSET);
434       }
435       // draw instructions straight line
436       drawStraightLine(motor, direction, duration);
437       ++i;
438     }
439     return 1;
440   }
441
442   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
443   //-----------------------------------------  Round Maze Maker
      ----------------------------------------------------//
444   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
445
446
447
448   // helper function returns the final angle of the circle instruction
449   int decodeFinalAngle(char in[6]) {
450     return (in[3] - '0')*100 + (in[4] - '0')*10 + (in[5] - '0'); // the -'0' is for converting from a
      char into an int
451   }
452
453   // helper function that returns the radius of the circle instruction
454   double decodeRadius(char in[6]) {
455     return in[2] - '0'; // the -'0' is for converting from a char into an int
456   }
457
458   // helper function that returns the distance of the radial line instruction
459   double decodeRadialDistance(char in[6]) {
460     return in[2] - '0'; // the -'0' is for converting from a char into an int
461   }
462
463   // 0 for clockwise
464   // 1 for counterclockwise
465   // 2 for in
```

```
466   // 3 for out
467   int decodeCircleInstruction(char in[6]) {
468     if ( in[1] == 'W') return 0;
469     if ( in[1] == 'C') return 1;
470     if ( in[1] == 'N') return 2;
471     if ( in[1] == 'O') return 3;
472     else         return 4; // this should never happen
473   }
474
475   // helper function sets the angle of the global variable
476   // called to change angle after motor is moved
477   // 0 for clockwise
478   // 1 for counterclockwise
479   int setAngle(int rotationDirection) {
480     if (rotationDirection == 0)
481       angle = angle + DEGREE_STEP;
482     if (rotationDirection == 1)
483       angle = angle - DEGREE_STEP;
484     // wrap around for the angles, makes sure it stays between
485     // 0 and 360
486     if (angle >= 360)
487       angle = angle - 360;
488     if (angle < 0)
489       angle = angle + 360;
490     return 1;
491   }
492
493   // helper function that draws a circle based on the circle instructions
494   int drawCircle(int instruction, double radius, int finalAngle) {
495     // function works by making straight lines in DEGREE_STEP's
496     // function continues to go until the final angle is reached
497     while (angle != finalAngle) {
498       // use trig to determine the angle of line drawn
499       // clockwise
500       double lineAngle = angle + DEGREE_STEP/2 + 90;
501       // counter clockwise angle
502       if (instruction == 1){
503         lineAngle = angle + (180-DEGREE_STEP)/2 + 180;
504       }
505       lineAngle = fmod(lineAngle,360);
506       // use trig to determine the distance of the line drawn
507       // d = 2*r*cos(beta)
508       // beta is the bigger angle of an isoleses triangle
509       // beta = (180 - DEGREE_STEP)/2
510       double lineDistance = 2 * radius * cos ( (180 - DEGREE_STEP) * PI / 360);
511       // draw said line
512       drawAngledLine(lineDistance, lineAngle);
513       // update the global variable angle after it's moved a degree
514       setAngle(instruction);
515     }
516     return 1;
517   }
518
519   // draws the maze based on array of instructions for circle maze
520   // first type of instructruction is a circle drawing
521   //     [0:1] CW for Clockwise, CC for counterclockwise
522   //     [2]   Radius
523   //     [3:5] Final Angle
524   // second type of instruction is a radial line drawing
525   //     [0:1] IN for traveling in, OU for traveling out
526   //     [2]   Distance to Travel
527   //     [3:5] No relavant information stored
528   int drawRoundMaze(char maze[][6], int arraylength) {
529     globalAngleCount = 0;
530     int i = 0; // keeps track of instruction number
531     while(i < arraylength - 1) {
532       int instruction = decodeCircleInstruction(maze[i]);
533       double radius = decodeRadius(maze[i]);
534       int finalAngle = decodeFinalAngle(maze[i]);
535       // if instruction is a rotation
536       if (instruction < 2) {
537         drawCircle(instruction, radius, finalAngle);
```

```
538       }
539       // if instruction is a radial movement
540       if (instruction > 1) {
541         double radialDistance = decodeRadialDistance(maze[i]);
542         double radialAngle = angle;
543         // if radially in, we want to switch direction
544         if (instruction == 2) {
545           // add 180 as we want the angle to be in the opposite direction as angle
546           radialAngle = fmod((radialAngle + 180), 360);
547         }
548         drawAngledLine(radialDistance, radialAngle);
549       }
550       // move to next instruction
551       ++i;
552     }
553     return 1;
554   }
555
556   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
557   //-------------------------------------  Bezier curve Drawing Functions
      ------------------------------------------------//
558   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
559
560   // helper function, takes in two positions and finds the angle between them
561   double angleFromPosition(double x1, double x2, double y1, double y2) {
562     double delta_x = x2-x1;
563     double delta_y = y2-y1;
564
565     // direction each component of the vector
566     int direction_x = 1;
567     int direction_y = 1;
568     if(delta_x < 0)
569       direction_x = 0;
570     if(delta_y < 0)
571       direction_y = 0;
572
573     // calculate triangle angle
574     double B_angle;
575     if (delta_x != 0) {
576       B_angle = 180*atan(fabs(delta_y)/fabs(delta_x))/PI;
577     } else {
578       B_angle = 90;
579     }
580     // translate into global angle of line
581     if (direction_x & direction_y)
582       B_angle = 180 - B_angle;
583     if (!direction_x & !direction_y)
584       B_angle = 360 - B_angle;
585     if (direction_x & !direction_y)
586       B_angle += 180;
587
588     return angle;
589   }
590
591   // helper function, takes in two positions and finds the distance bewteen them
592   double distanceFromPosition(double x1, double x2, double y1, double y2) {
593     double delta_x = x2-x1;
594     double delta_y = y2-y1;
595     // pythagoras
596     return sqrt(delta_x*delta_x + delta_y*delta_y);
597   }
598
599   // giving four points, draws a curved line that starts at point 0 and ends at point 3. When leaving
      point 0,
600   // it travels in the direction of point 1. When arriving at point 3, it is traveling from the direction
      of point 2.
601   // this function allows the assembly to draw any curved
602   int drawBezierCurve(double x0, double y0, double x1, double y1, double x2, double y2, double x3, double
      y3) {
603     for(int t = 0; t < 10; ++t) {
```

```
604       // taken from formula used to calculate a cubic bezier curve
605       // it's essentially a weighted average of the 3 points based on t. As we varry t, we go through the
      curve
606       double B_x      = x0*pow((1-(t/10)), 3) + 3*x1*pow((t/10)*(1-(t/10)),2) +
      3*x2*(1-(t/10))*pow((t/10),2) + x3*pow((t/10),3);
607       double B_y      = y0*pow((1-(t/10)), 3) + 3*y1*pow((t/10)*(1-(t/10)),2) +
      3*y2*(1-(t/10))*pow((t/10),2) + y3*pow((t/10),3);
608
609       // figure out what the next position in the curve is
610       int n = t + 1;
611       double B_x_next = x0*pow((1-(n/10)), 3) + 3*x1*pow((n/10)*(1-(n/10)),2) +
      3*x2*(1-(n/10))*pow((n/10),2) + x3*pow((n/10),3);
612       double B_y_next = y0*pow((1-(n/10)), 3) + 3*y1*pow((n/10)*(1-(n/10)),2) +
      3*y2*(1-(n/10))*pow((n/10),2) + y3*pow((n/10),3);
613
614       // use this information to determine
615       double lineAngle = angleFromPosition(B_x, B_x_next, B_y, B_y_next);
616       double distance = distanceFromPosition(B_x, B_x_next, B_y, B_y_next);
617
618       drawAngledLine(distance, lineAngle);
619     }
620     return 1;
621   }
622
623   // Draws a Series of Bezier Curves
624   int drawPath(double path[], int arrayLength) {
625     // so every Bezier Curve needs a starting and ending point. But for a continuous path, the ending
      point of one Bezier Curve,
626     // is the starting point of the next Bezier Curve. Each point is two elements of path[]. Thus for
      every instruction, the i
627     // needs to be incremented by 6, (aka for 3 points)
628     for(int i = 0; i < arrayLength-1; i = i + 6) {
629       drawBezierCurve(path[i], path[i+1], path[i+2], path[i+3], path[i+4], path[i+5], path[i+6],
      path[i+7]);
630     }
631     return 1;
632   }
633
634   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
635   //-----------------------------------------    Main Functions
      ------------------------------------------------//
636   ////////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////
637
638   // Main Function Runs loop to check for input
639   int main(void) {
640     // initialize clock
641     samInit();
642     // initialize pins
643     pioInit();
644     // initialize clock
645     tcDelayInit();
646     // initialize pins that are used
647     pinInit();
648
649     // while loop that checks for button presses
650     while(1) {
651       // Draws Easy Maze
652       if (pioDigitalRead(BUTTON1_IN)== PIO_HIGH) {
653         drawStraightMaze(maze0, 47);
654       }
655       // Draws Medium Maze
656       if (pioDigitalRead(BUTTON2_IN)== PIO_HIGH) {
657         drawStraightMaze(maze1, 103);
658       }
659       // Draws Hard Maze
660       if (pioDigitalRead(BUTTON3_IN)== PIO_HIGH) {
661         drawStraightMaze(maze2, 129);
662       }
663       // Draws Circle Maze
664       if (pioDigitalRead(BUTTON4_IN) == PIO_HIGH) {
```

```
665           drawRoundMaze(circletest1, 65);
666       }
667       // Draws Long Maze
668       if (pioDigitalRead(BUTTON5_IN) == PIO_HIGH) {
669         // shrinks the size of the maze drawing
670         INITIAL_OFFSET = 500;
671         DISTANCE_COEFF = 500;
672 
673         drawStraightMaze(mazeLong, 495);
674 
675         // return offsets to orriginal values
676         INITIAL_OFFSET = 650;
677         DISTANCE_COEFF = 650;
678       }
679     }
680   }
681 
```

```
1  ////////////////////////////////////////////////
2  // Motor Control Function for dual stepper motors
3  // HMC E155 Dec 7th 2018
4  // dtowersm@gmail.com
5  // written by Daniel Torres and Martha Gao
6  ////////////////////////////////////////////////
7
8  module motor_control(input  logic motor, direction, clk,
9                       output logic [0:3] a_coil, // controls left motor
10                      output logic [0:3] b_coil); // controls right motor
11
12
13      // 0 for left motor, 1 for right motor
14      motor_fsm  left_motor(clk, ~motor, direction, a_coil);
15      motor_fsm right_motor(clk,  motor, direction, b_coil);
16 endmodule
17
18
19 // clk turns on the motor allowing it to go to next state
20 // direction tells the states which direction to go in
21 // 0 means clockwise, 1 means counter clockwise
22 module motor_fsm(input  logic clk, motor, direction,
23                  output logic [0:3] coil);
24    // define states of FSM
25    typedef enum logic[2:0]{S0, S1, S2, S3, S4, S5, S6, S7} statetypes;
26    statetypes state, nextstate;
27
28    // only one motor is moved at a time,
29    // so the enable is crucial
30    always_ff@(posedge clk)
31       if (motor) state <= nextstate;
32
33    // next state logic
34    // direction being 1 makes the motor move
35    // in the counter clockwise direction
36    always_comb
37       case(state)
38          S0: if(direction)    nextstate <= S7;
39             else              nextstate <= S1;
40          S1: if(direction)    nextstate <= S0;
41             else              nextstate <= S2;
42          S2: if(direction)    nextstate <= S1;
43             else              nextstate <= S3;
44          S3: if(direction)    nextstate <= S2;
45             else              nextstate <= S4;
46          S4: if(direction)    nextstate <= S3;
47             else              nextstate <= S5;
48          S5: if(direction)    nextstate <= S4;
49             else              nextstate <= S6;
50          S6: if(direction)    nextstate <= S5;
51             else              nextstate <= S7;
52          S7: if(direction)    nextstate <= S6;
53             else              nextstate <= S0;
54          default:             nextstate <= S0;
55       endcase
56    // assign logic
57    // turn on the correct coils depending on the state
58    assign coil[0] = (state == S5) || (state == S6) || (state == S7);
59    assign coil[1] = (state == S3) || (state == S4) || (state == S5);
60    assign coil[2] = (state == S1) || (state == S2) || (state == S3);
61    assign coil[3] = (state == S7) || (state == S0) || (state == S1);
62 endmodule
```