Milestone 4

EEE3099S Mechatronics Design

Group 11 Robot Name: Aubrey Mamabusa

Introduction

The task at hand was creating a line-following robot which solved a maze. The robot needed to interface with a circuit that was designed to indicate if the robot was detecting red or green light, and also included buttons for communicating with the robot. The maze that the robot had to solve included lines for it to follow as well as obstacles like river crossings, T-junctions, 4-way junctions, left or right turns, and dead ends. The robot needed to differentiate between these aspects of the maze using sensors which detect black or white (whether it was on a black line or not) and needed to account for any differences between the obstacles or any overcorrections made.

System Level Design

Junction Decision Making:

It was decided that upon reaching a junction, the robot would always make a left turn if one was available. This decision was made after inspecting the mazes that the robot was being tested on and finding that always making the left turn was faster than always making a right turn.

State Flow Instead of PID:

The decision to use stateflow to implement the maze solving algorithm instead of some sort of PID controller was made to allow for ease of debugging and tuning as the stateflow representation is more aligned with our current understanding of coding.

Placement of Light Sensing Circuitry:

The light sensing circuitry was placed facing the back of the robot as per the milestone brief, we could choose where the traffic light was placed and would be facing. Thus, it made sense to make the light sensing circuit face backwards as this freed up room for the microcontroller itself.

Interrupts for Reading Encoders:

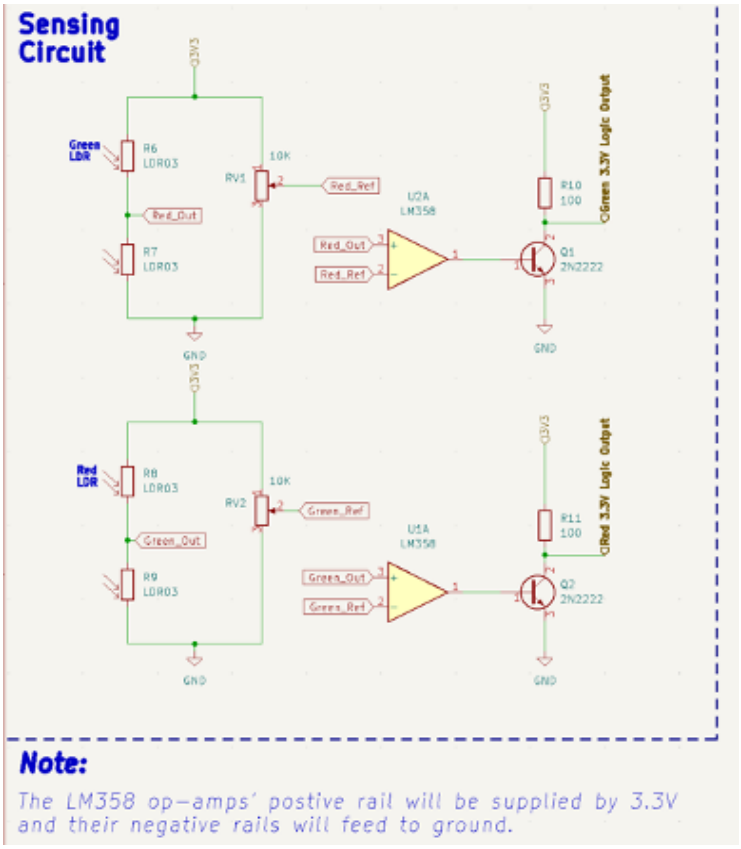
Interrupts were used to read the encoder values instead of normal function calls as this way the program would not miss encoder ticks or have to wait for the encoder value to be read.

Electrical Design

Light Sensor Design:

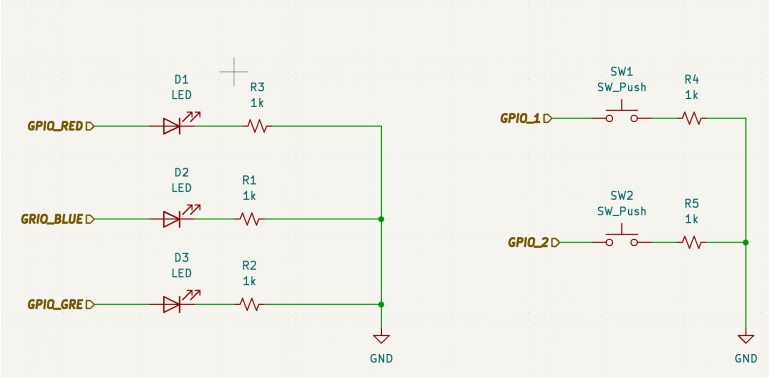
LDR's sensitive to red and green light are created by covering them with red and green film. Each LDR is then placed in a voltage divider with another LDR that is not covered with any film. These LDR's were chosen to ensure that the output of the voltage divider is based on the ratio between them and thus will not change dramatically due to ambient light. The output of these voltage dividers is compared to the voltage output of a potentiometer using an op-amp. The potentiometer will be used to calibrate the sensor to account for changes in ambient light. The op-amp will be operated in saturation.

The output of the op-amp is fed into the base on an NPN transistor to buffer the output signal and ensure that the 3.3V logic level requirement is satisfied.



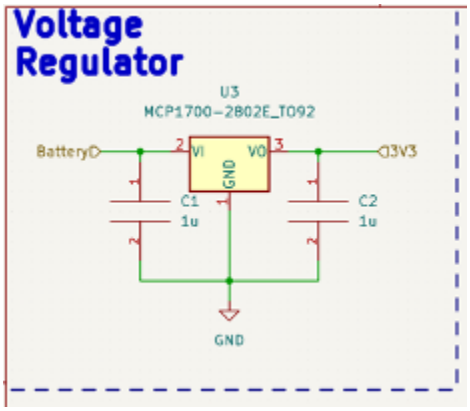
Communication Circuitry:

10kΩ pull-down resistors were used to ensure that when the pushbutton is not pressed the logic output levels of the pushbuttons is 0V and 3.3V when pressed.
1kΩ resistors are put in series with the state LEDs to ensure the current is limited to preserve the LEDs.



Power Circuitry:

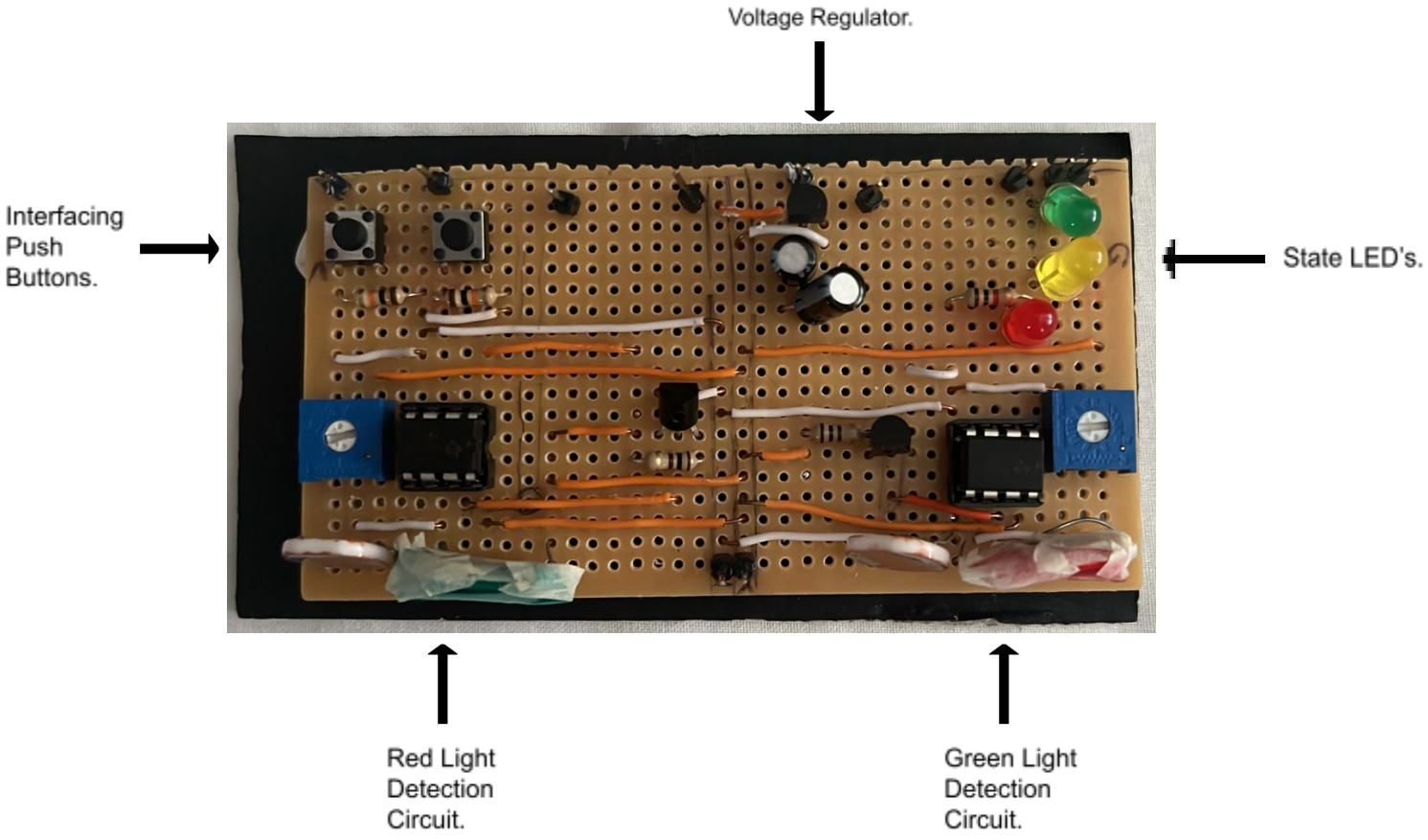
A MCP1702 3.3V voltage regulator was used. Decoupling capacitors were added to both the input and output to better improve voltage stability. The MCP1702 was chosen as it is rated for voltages between 2V and 13.2V which includes the desired voltage range between 5V and 8.4V.



Power Calculations:

Component	Current [A]	Voltage [V]	Power [W]
Sensing Circuit	0.015	5	0.075
Motors and Driver	0.6	5	3
Total	0.615	5	3.075

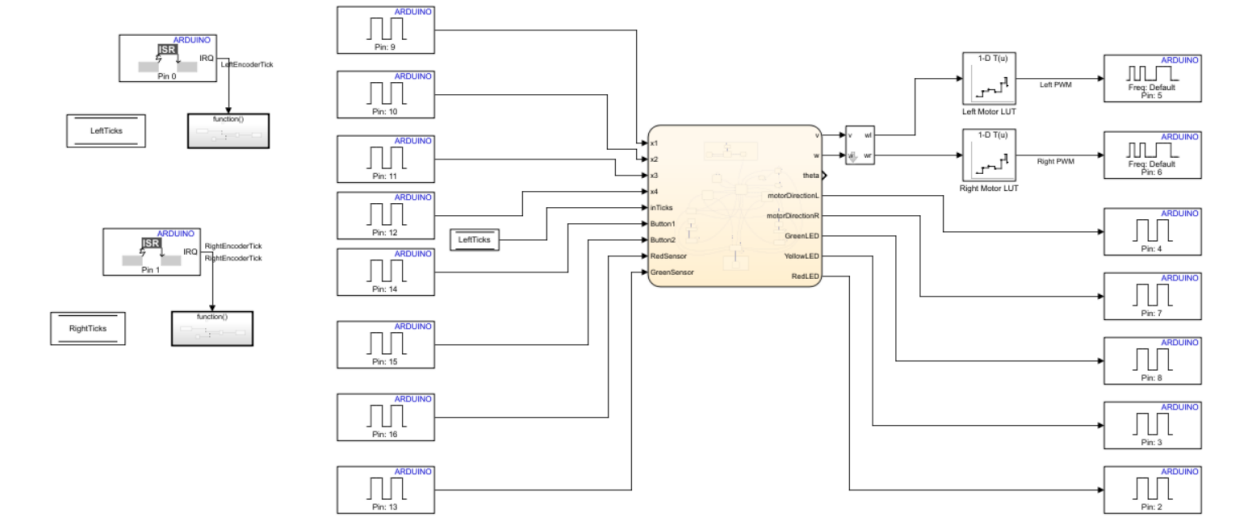
Actual Veroboard Layout:



Robot:

The robot design was not ours to take credit for. The robot was built using a pre-existing kit from DFRobot.

Board setup and pinout



Microcontroller Resource Allocation:

Pin Number	Type	Associated Variable	Hardware Association to StateFlow
9	Input	x1	Leftmost sensor
10	Input	x2	Inner left sensor
11	Input	x3	Inner right sensor
12	Input	x4	Rightmost sensor
13	Input	GreenSensor	Pin showing detection of green light
14	Input	Button1	Goes high if button 1 pressed
15	Input	Button2	Goes high if button 2 pressed
16	Input	RedSensor	Pin showing detection of red light

2	Output	RedLED	Toggles the state of the red LED on the veroboard
3	Output	YellowLED	Toggles the state of the yellow LED on the veroboard
4	Output	motorDirectionL	Toggles direction of the motor (1 for fwd, 0 for back)
5	Output	RightPWM (v and w)	Gives motor speed based on v and w
6	Output	LeftPWM (v and w)	Gives motor speed based on v and w
7	Output	motorDirectionR	Toggles direction of the motor (1 for fwd, 0 for back)
8	Output	GreenLED	Toggles the state of the green LED on the veroboard

Motion control design, implementation and results:

Design and Implementation

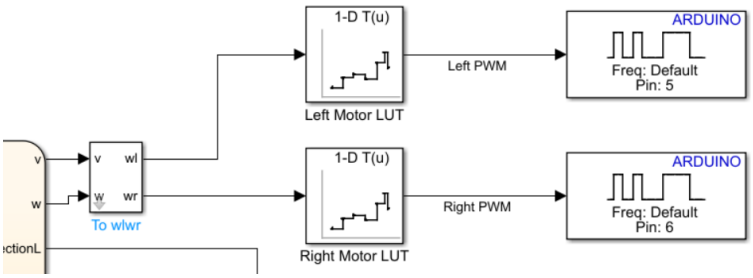
The robot's motion was controlled in MATLAB using a *To WLWR* block which takes in as inputs a desired linear velocity *v* and a desired rotational velocity *w* and converts these values to left right wheel rotational velocities *wl* and *wr.* The block also stores the wheel radius and the axle length which are used to convert the inputs to the outputs using the following formulas:

```
w_L  = (v - w * axleLength/2)/wheelRadius
w_R = (v + w * axleLength/2)/wheelRadius
```

These *wl* and *wr* values were then fed into lookup tables designed for the motors used in the robot. A snippet of these tables is shown below:

1	**-12.51**	-127
2	**-12.38**	-126
3	**-12.24**	-125
4	**-12.10**	-124
5	**-11.97**	-123
6	**-11.83**	-122
7	**-11.70**	-121
8	**-11.56**	-120
9	**-11.43**	-119
10	**-11.29**	-118
11	**-11.16**	-117

These values were then fed into PWM output blocks and output through Pins 5 and 6 on the Arduino board to control each motor. The direction of the motors was controlled using digital outputs at Pins 4 and 7 with a 1 causing the motor to rotate forwards and 0 causing the motor to rotate backwards. The overall connection layout looked as follows:

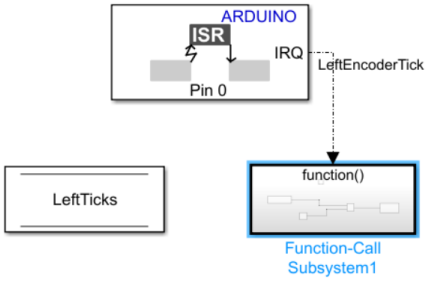


The following functions were used to control the motion of the robot, based on certain states of the line sensors:

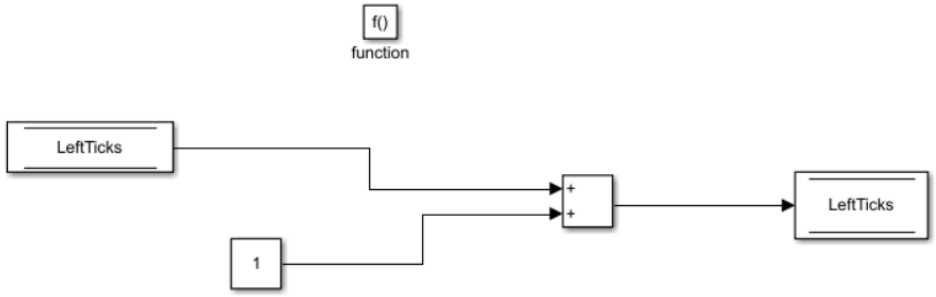
Block Name in Stateflow	**Effect on Robot**	**V and W values**	**Pin Values**
MovingForward	The motor moves in a straight line with a constant velocity. Motors move the robot forward with no rotational velocity.	v = 0.06 w = 0	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
TurningRight	Used to rotate the robot slightly to the right whilst continuing to move forward.	v = 0.04 w = 0.5	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
TurningLeft	Used to rotate the robot slightly to the left whilst continuing to move forward.	v = 0.04 w = -0.5	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
ShiftRight	Used to rotate the robot slightly to the right.	v = 0 w = 0.4	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
ShiftLeft	Used to rotate the robot slightly to the left.	v = 0 w = -0.4	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
NinetyTurnRight	Used to turn the robot 90 degrees to the right by setting w high for a set number of encoder ticks.	v = 0 w = 1	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
NinetyTurnLeft	Used to turn the robot 90 degrees to the left by setting w high for a set number of encoder ticks	v = 0 w = -1	Output a 1 to Pins 4 and 7 to make motors rotate forwards.

Delay	Keeps V and W the same as the previous state (NinetyTurnLeft/Right) for a certain number of encoder ticks.	v = vprev w = wprev	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
whileLoopRot	Keeps V and W the same as the previous state (NinetyTurnLeft/Right) until certain line sensor conditions are met.	v = vprev w = wprev	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
Delay3	Move the robot forward slowly and slightly for 6 motor encoder ticks until the condition of the sensors changes. If no change, the robot has reached GameOver.	v = 0.04 w = 0	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
WaitForLine	The robot moves forward with a slight correcting rotational velocity (to account for any shift of the right wheel, found through trial-and-error) until the river is crossed and a line is detected.	v = 0.04 w = 0.03	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
CorrectRight	Catch-condition: catches the line on the right-most sensor: Rotate the robot clockwise while still moving forward, until it can move straight.	v = 0.04 w = 0.5	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
CorrectLeft	Catch-condition: catches the line on the left-most sensor: Rotate the robot counter-clockwise while still moving forward, until it can move straight.	v = 0.04 w = -0.5	Output a 1 to Pins 4 and 7 to make motors rotate forwards.
OneEightyTurn	The left motor moves backwards whilst the right motor moves forwards, both at the same velocity, so that the robot rotates on its central axis until the black line is found.	v = 0.07 w = 0	Output 1 to pin 4 (right) and 0 to pin 7 (left) to make the motors rotate in opposite directions. Left motor will move backwards while Right will move forwards.
GameOver	The robot stops moving as it reaches the end of the maze	v = 0 w = 0	—

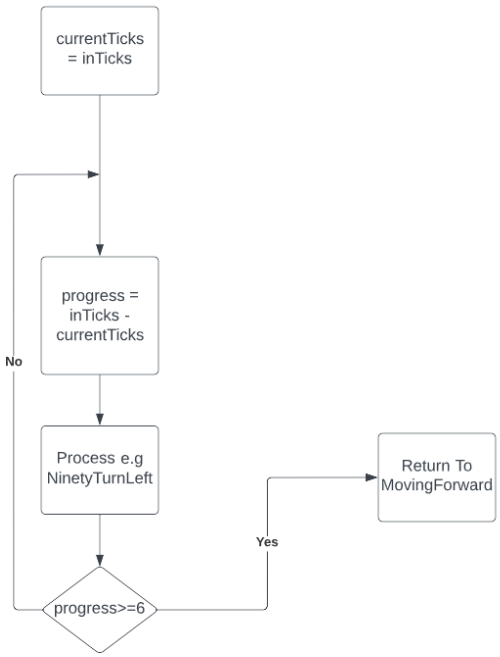
The delays were created by measuring the encoder ticks. The encoder ticks were measured using an interrupt block connected to Pin 0 which triggered an interrupt whenever the encoder pin went high. This interrupt called a function which incremented a RightTicks and LeftTicks variable by 1. The Matlab layout looked as follows:



The above interrupt called the following function:



The exact same function was implemented for the right encoder using a RightTicks variable. These encoder ticks were used to create a delay as follows:



In the above diagram inTicks is constantly reading the value of the leftTicks variable. currentTicks is set to the inTicks value at a specific instance in time and progress measures how many ticks have passed since currentTicks was set.

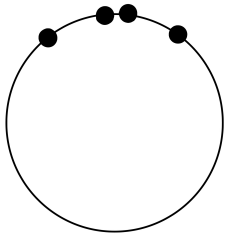
Results:

The motors behaved very much as expected and the functions worked as planned. The only functions that could have been improved upon were the NinetyTurnLeft/Right functions because these functions caused the motors to turn on the axis of one of the wheels rather than on the center axis of the robot. This often caused the robot to turn far off of the line making it taking unnecessarily long to make the turn. This could be improved upon by instead implementing an algorithm similar to the one used in OneEightyTurn and causing one motor to rotate backwards and the other to rotate forwards.

Line Sensing Design, Implementation and Results:

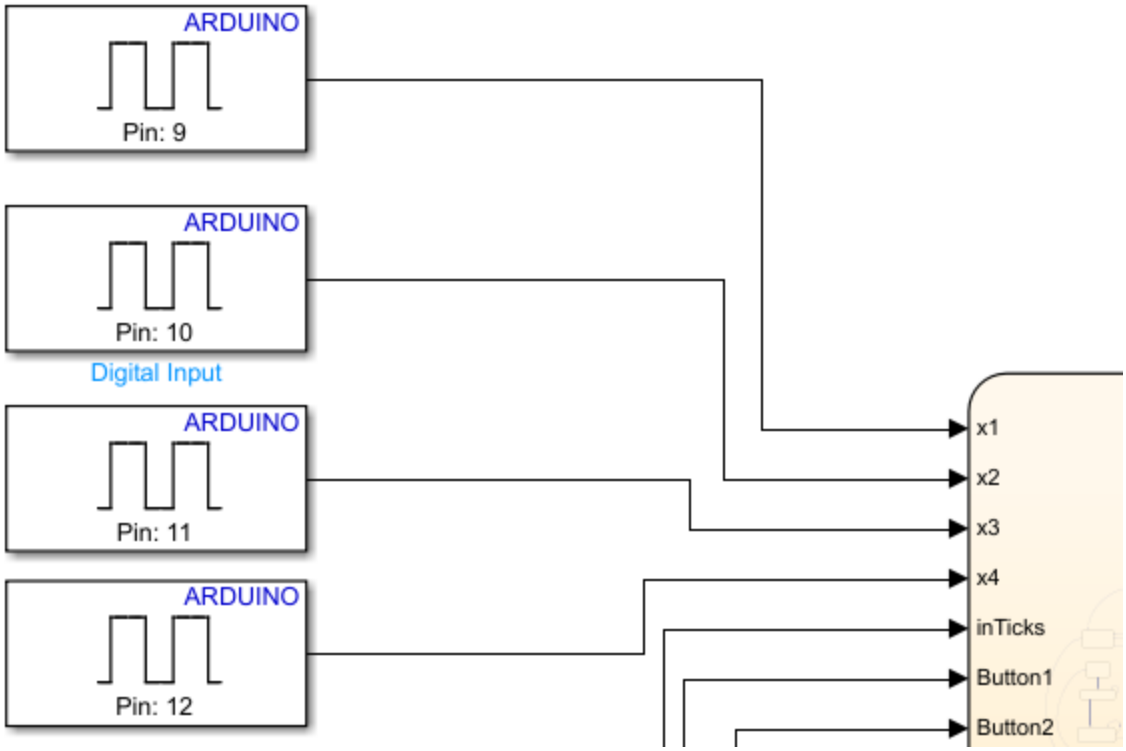
Design of Hardware and Software:

The line sensors were placed on the robot in the following manner:

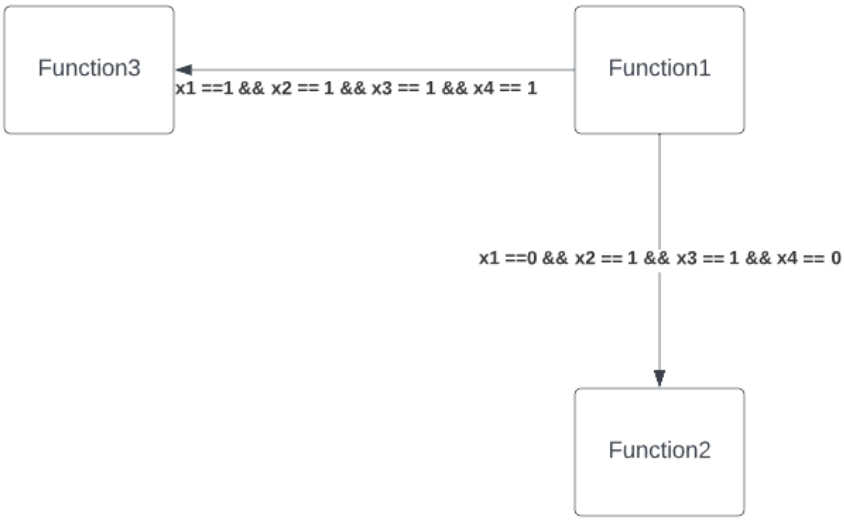


This was done so that the center two could always track the line that the robot was following and the outer two were used when checking for turns and river crossings and junctions.

In terms of software, the line sensors were connected to pins 9, 10, 11 and 12 and were read into Matlab using arduino digital input blocks. These blocks were then fed into stateflow and each sensor was assigned to a variable named x1, x2, x3 and x4 respectively.



The status of each sensor was used to determine which function block in stateflow the program entered. A basic diagram showing how this worked is shown below:



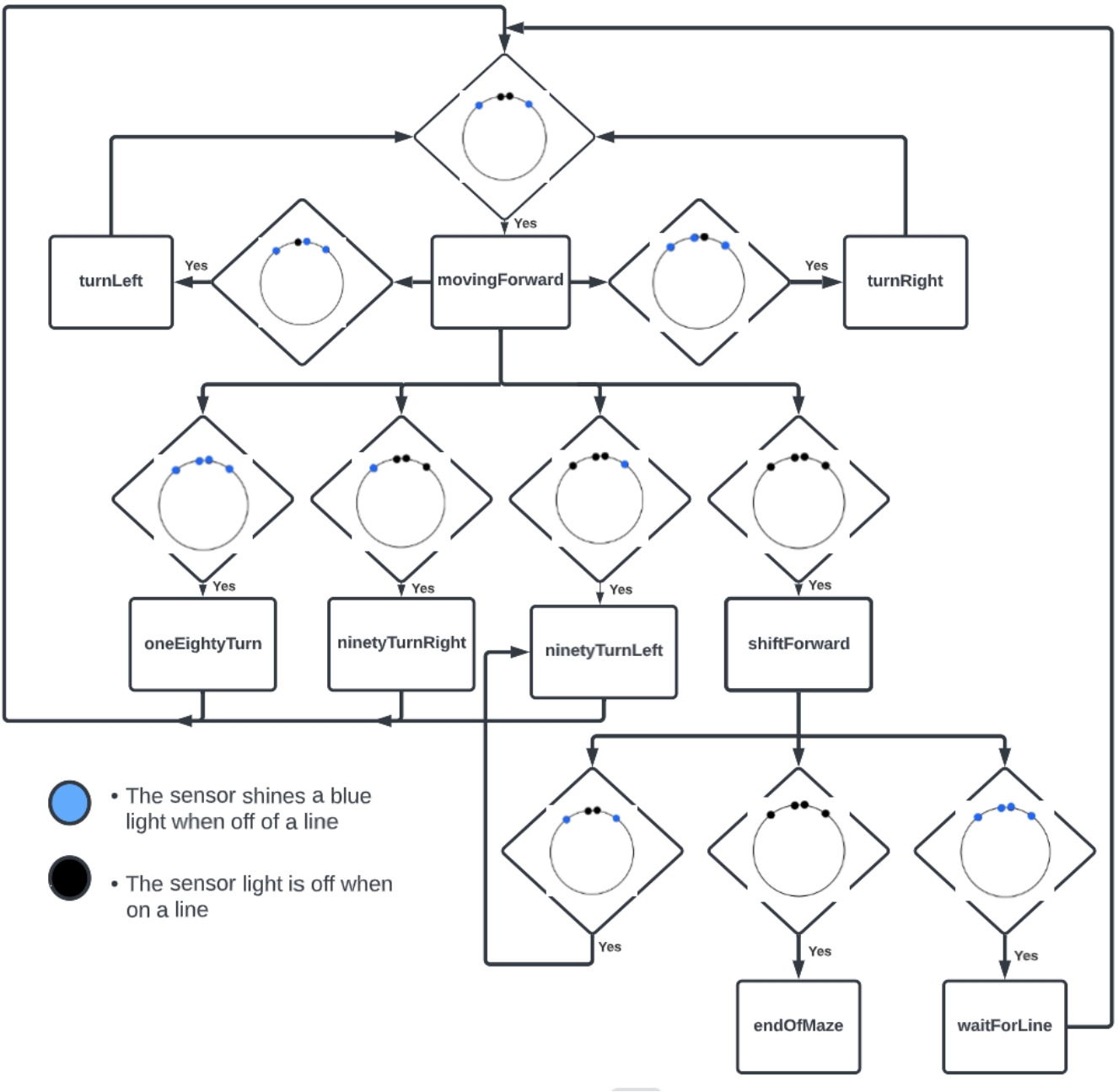
Interrupts were not used and instead a real time structure in stateflow was used to trigger a function because our algorithm relied on the program staying in a particular state until the line sensor status changed instead of just triggering a function once when a line sensor condition was met.

Treasure Maze solving algorithm design:

Algorithm Description:

1. LED and light sensing: robot senses light when button 1 is pressed and the robot starts the maze when button 2 is pressed, if green light is sensed.
2. Once the robot has started the maze, if the middle sensors are detecting a line, the robot will stay in the **MoveForward** state.
3. If either of the middle sensors go off the line, it will either veer left or veer right in states **TurnLeft** or **TurnRight**.
4. If the robot detected 3 sensors on a line, it triggered the state **NinetyTurnLeft** or **NinetyTurnRight** depending on which outer sensor was on the black line. **whileLoopRot** (and **whileLoopRot1**) ensures that the robot stays in a turning condition until the two middle sensors hit a black line.
5. If all 4 sensors were detected to be on a line, the robot triggered a **Delay** and moved forward slightly. This allowed testing of either the river crossing or a 4-way crossroads.
6. If the robot shifted forward and detected white with all 4 sensors, the robot moved into its **River Crossing state** called **WaitForLine**. The states **ShiftLeft** and **ShiftRight** were used for correction within a river crossing to make sure we hit the line straight.
7. Otherwise, if black was detected by the middle 2 and white on the outer sensors after shifting forward slightly, the robot was at a 4-way junction and defaulted to a left turn (output 4 of Delay3 state), until the 2 middle sensors hit a line again.
8. If after shifting forward always sensors detected black, the robot detects the end of the maze and stops moving.
9. A metric called **progress** was used as a delay (instead of a time delay) using encoder ticks. The **progress** was the number of ticks since one condition was met and was used as a delay before moving into another state. The number of ticks passed was used as a condition to be met for the river crossing, the 4-way junction, and the ending block.
10. The ending block counted the most number of encoder ticks (progress) before making the robot stop (progress>=7 while all sensors are on black).
11. If all sensors detected white after being on a straight black line (in **MovingForward**), the **oneEightyTurn** state was triggered which rotated the robot on its central axis until the middle 2 sensors hit a black line again, after which it moved back into **MovingForward**.

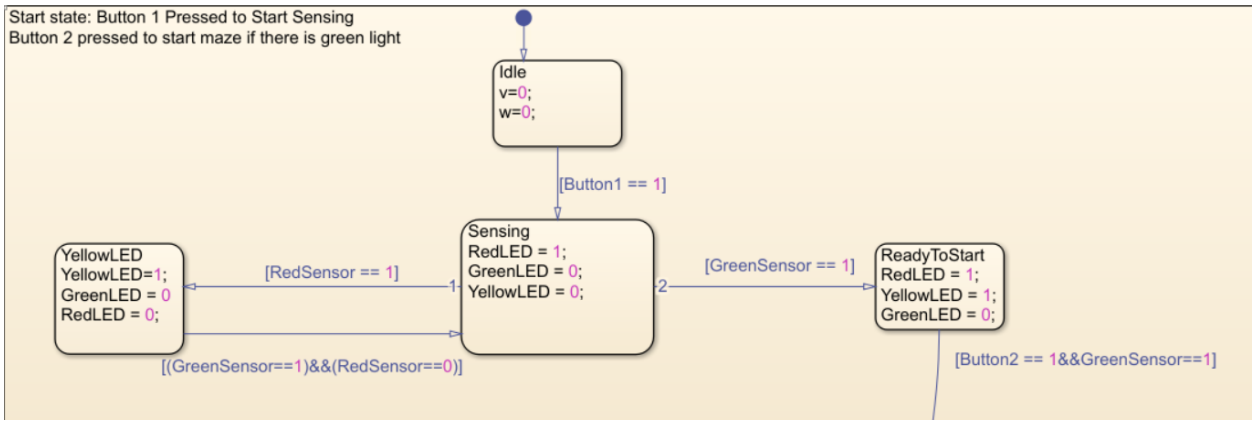
Flow Chart of Maze Solving Algorithm:



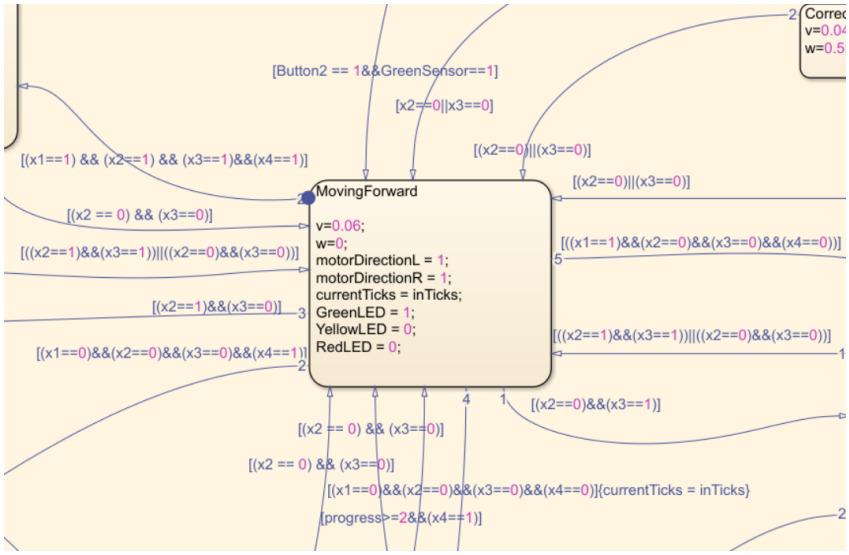
Maze Solving Algorithm Implementation:

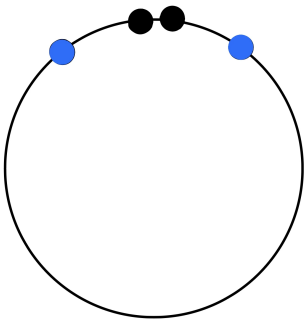
The algorithm was implemented using state flow logic. It resembles the algorithm description above with the addition of certain delays to help improve robustness in the physical implementation.

LED and Light Sensing Logic



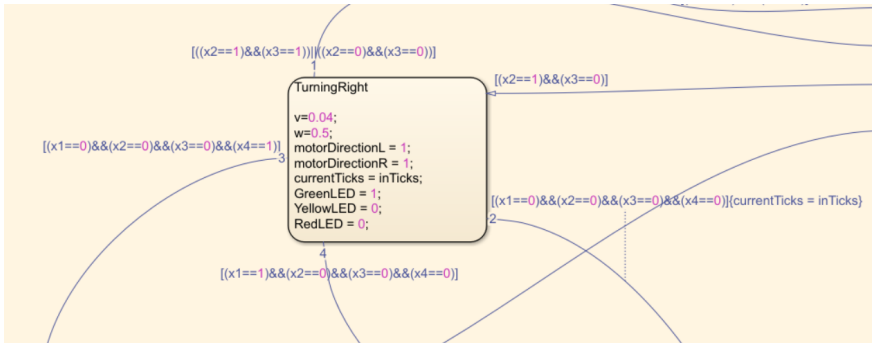
MovingForward State entry and exit conditions

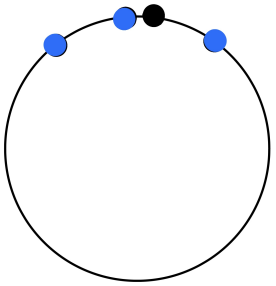




Sensor configuration for moving straight.

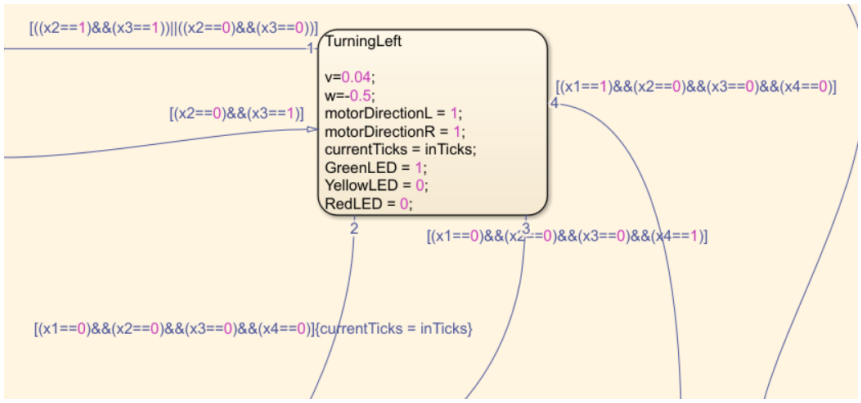
TurningRight State entry and exit conditions

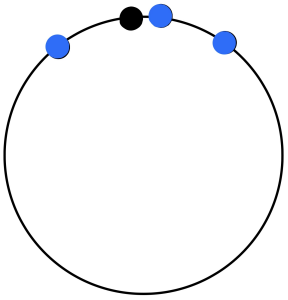




Sensor configuration for veering right.

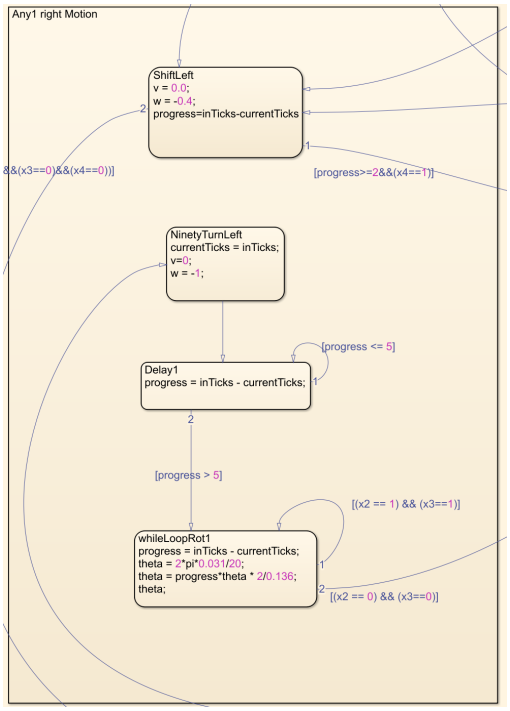
TurningLeft State entry and exit conditions

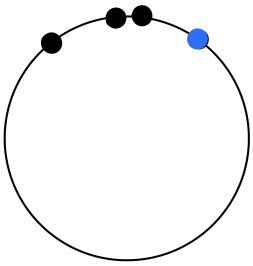




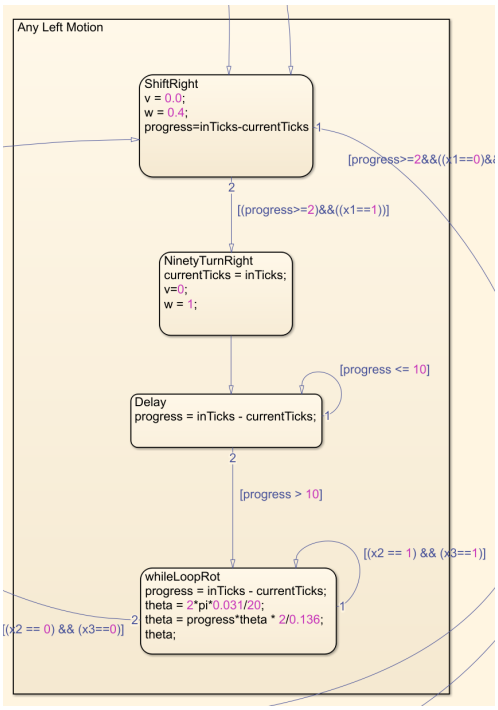
Sensor configuration for veering left.

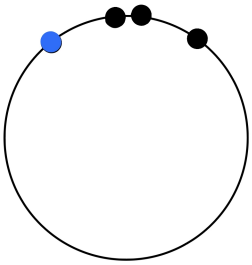
Motion in either right or left directions, and correction of motion direction





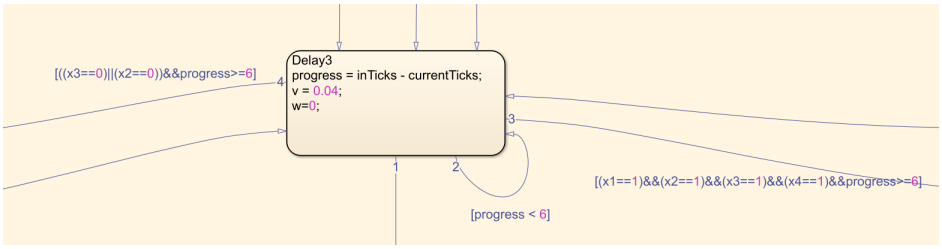
Sensor configuration for turning 90 degrees to the left.

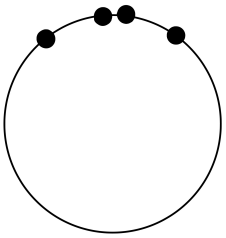




Sensor configuration for turning 90 degrees to the right.

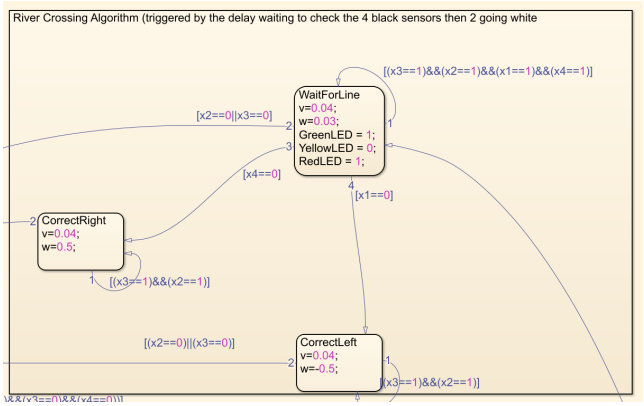
Delay for when 4 sensors are black and exit conditions

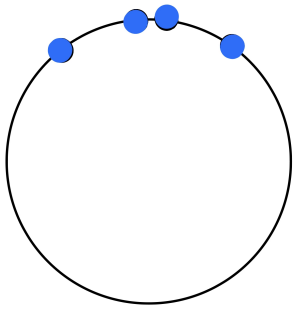




Sensor configuration for triggering shift forward

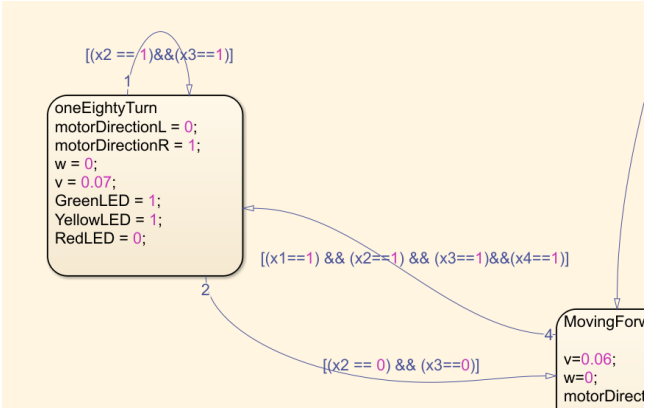
River crossing state and exit conditions

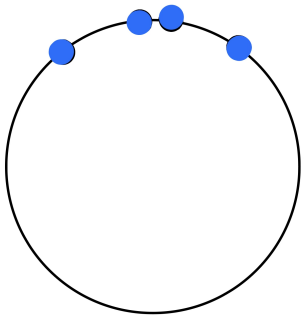




Sensor configuration for river crossing after shifting forward.

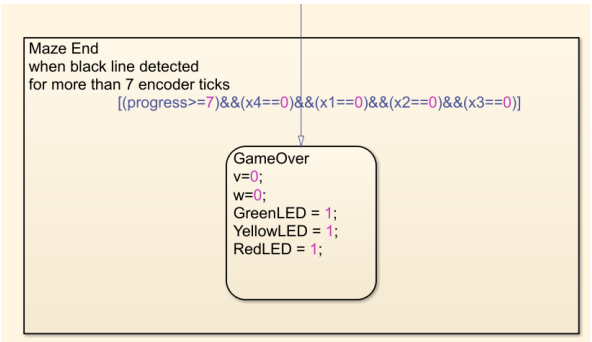
oneEightyTurn state entry and exit conditions

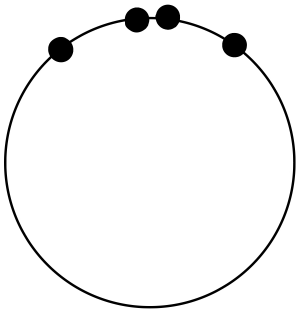




Sensor configuration for 180 degree turn if in moving forward state.

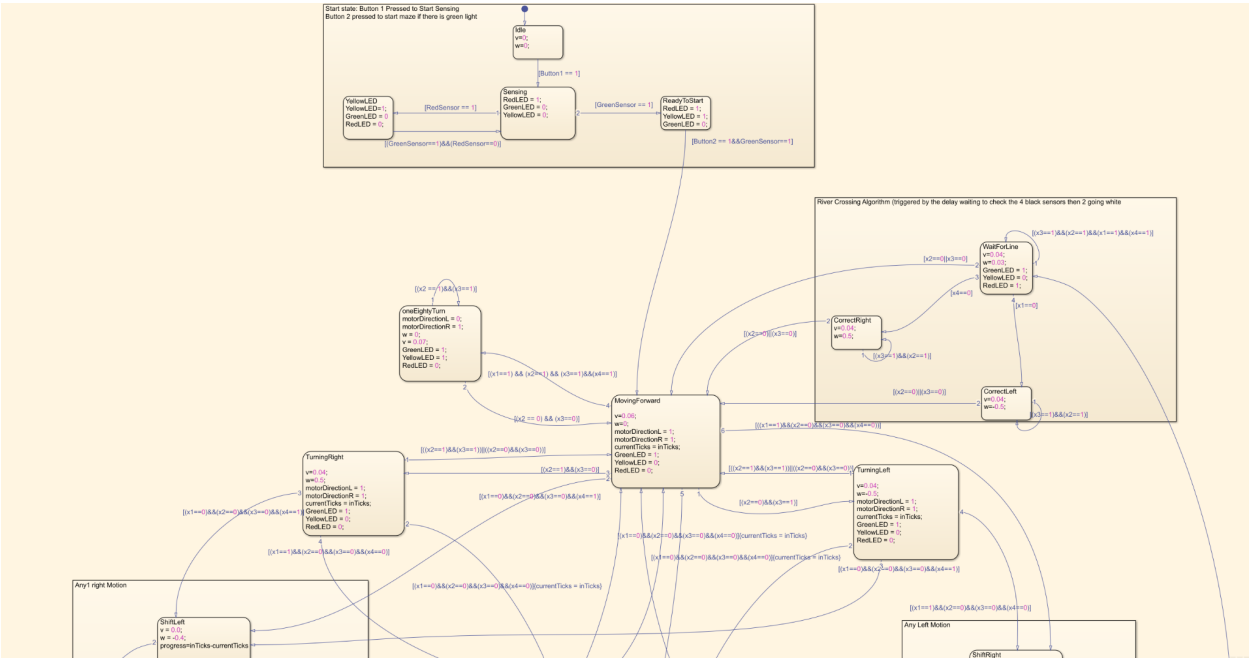
End of Maze

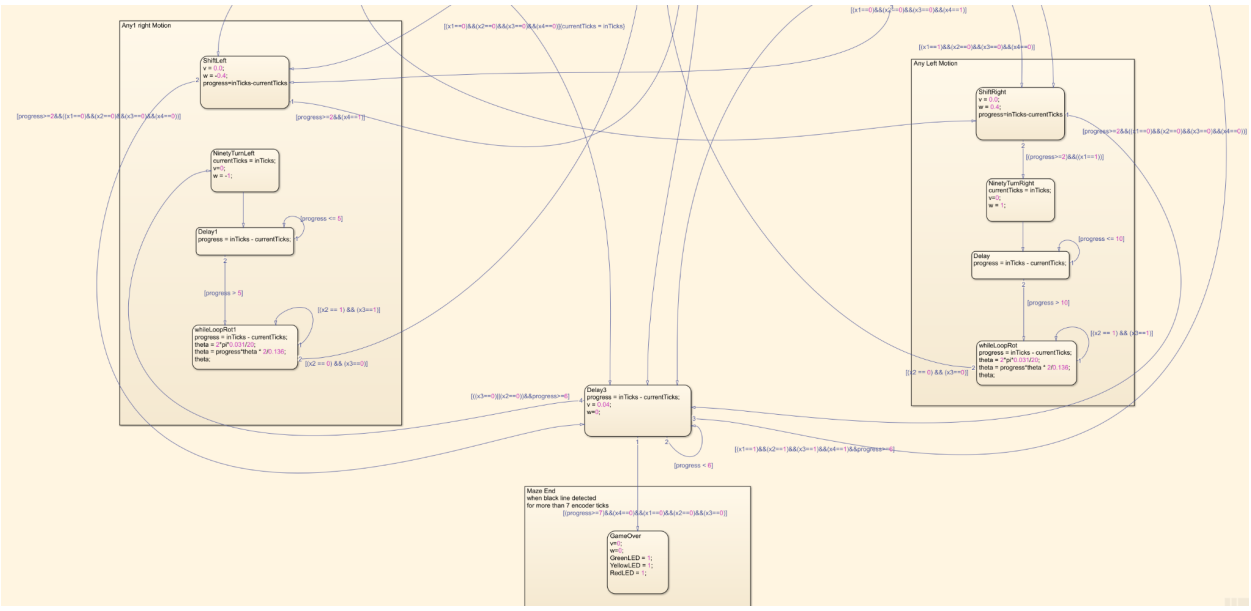




Sensor configuration for end of maze after shifting forward.

Full Matlab Screenshots





Maze Solving Algorithm and Implementation Results

In practice the maze solving algorithm detailed above behaved as expected. It correctly and accurately followed a line, detected right angled turns, cross-roads, river crossings and the end of the maze. For the most part, the robot correctly performed a 180 degree turn when required, however, it was found that if the line it was following curved sharply without being a right angled turn, the robot sometimes interpreted this as a 180 degree turn. It is hypothesized that this behavior could be corrected by tuning the forward and rotational velocities of the robot to allow for faster correction.

Conclusion

The robot solved the final maze as expected, with only one instance where it needed intervention because its rotational velocity was not high enough to overcome a sharp bend in the line. The robot handled dead ends, 90° corners/turns, 4-way junctions, river crossings and general line-following perfectly and finished the maze in a time of 2 minutes and 51 seconds (2:51). The robot got stuck at one river crossing which it treated as a corner, but managed to find its way back to repeating the river-crossing and executing it perfectly, moving towards the end of the maze speedily. The implementation of the general motion control, line-following and obstacle handling using MATLAB Simulink allowed for ease of integration of logic and motion control with the hardware. The only downside to using the DFRobot Arduino Leonardo board was that we were unable to deploy code to the board and had to run the code whilst the board was connected to a laptop.