Final Milestone Report

Micromouse Project Report

Prepared by:
Zuhayr Halday (HLDZUH001), Kavya Kaushik (KSHKAV001)

Prepared for:
EEE3097S
Department of Electrical Engineering
University of Cape Town

October 22, 2024

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

October 22, 2024

Partner 1 Signature

Date

Partner 2 Signature

Contents

Chapter 1

Executive Summary

1.1 Introduction

The project focuses on building a fully functional micromouse robot that can sense its surroundings and thereby automatically traverse through a maze using the shortest and fastest route possible. The project is a continuation from the EEE3088F course which focused on designing and testing the hardware power and sensing subsystems of the micromouse. In the current EEE3097S course focused on assembling the micromouse's hardware components, analysing the how the various components interact with each other and programming the micromouse to navigate the maze efficiently.

The motherboard serves as the main mount of the micromouse robot and enables the connection between the power and sensing subsystem with the remaining components. These connections run the motors, which in turn drive the wheels of the mouse. The power subsystem is designed to supply uninterrupted power to drive the micromouse's motors and to charge to battery using an external power source. The sensing subsystem is designed to detect the walls of the maze and relays this information to the microcontroller to navigate the maze.

The micromouse maze is built using wood and the lanes are marked with black electrical tape placed on all paths through the maze. The mouse is required to follow the tape line, turn at path intersections and reach the center of the maze, thus solving it.

Before the programming, the power board, processor board, motherboard, programmer adapter, sensor board and the motors were all assembled through soldering to build the micromouse robot.

This report discusses the design, implementation and testing of the micromouse robot, highlighting the navigation logic implemented, maze solving algorithms and path optimisation. It also explores the challenges encountered in the process including motor control issues and sensor accuracy while proposing potential enhancements for the micrmouse's performance.

1.2 Requirements

The project requirements were divided between four milestones. The requirements of each milestone are detailed in Table I

TABLE I Micromouse Requirements (Milestone 1-4)

Milestone	**Objective**	**Key Achievements**
Milestone 1	Hardware Design and Testing	- Analysed the functionality and configuration of the sensing and power subsystem.
		- Discussed the interfacing of the power and sensing subsystem with each other and with the remaining motherboard and microcontroller
Milestone 2	Sensor Calibration and Basic Navigation	- Initialization routines for motors and sensors developed.
		- Partial sensor calibration; spatial awareness readings noted.

Milestone	Objective	Key Requirements
		- Line-following mechanism implemented; micromouse can hold line direction and adjust for left/right wiggle.
		- Wall detection functionality implemented using sensor data.
Milestone 3	Movement, Decision-Making, and Turning	- Decision-making algorithms for micromouse movement and turns
		- Basic maneuvering through maze enabled.
Milestone 4	Path Optimization and Maze Navigation	- Minimize errors while navigating the maze.
		- Maximize the time it takes to complete the maze.

1.3 Simulink Model

A MATLAB Simulink System was provided by the course, which models the STM32 Processor, motors, Power Subsystem, and the Sensing Subsystem all in one Simulink model. This system was used to program the micromouse, as the graphical environment of Simulink allowed for easier and more intuitive development than programming the microcontroller directly with C.

As can be seen in Figure 1.1, the Simulink files configured the link between hardware and software modelling and configured the GPIO, ADC and all other pin ports of the LEDs, motors and pushbutton. The Stateflow program template provided the logic to control the logic of all other models and blocks in the Simulink file.

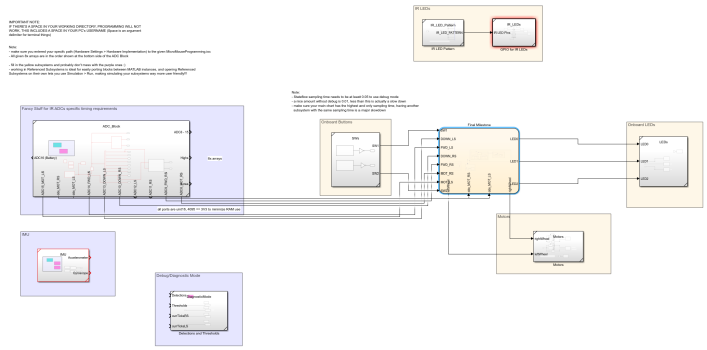

Figure 1.1 MATLAB Simulink Template for all Subsystem Configuration/Programming

1.4 Limitations and Adjustments

The micromouse robot was limited by the hardware components provided to assemble the micromouse. It was realised that the right and left sensing LEDs were placed on the center tab of the sensing board, which meant they could not accurately measure their distance from the maze walls as they were positioned too far from the walls. These LEDs were therefore not used in the navigation design.

The values to start the motors and direct them backwards or forwards were set different to the standard values of zero(for stopping), positive(forward movement) and negative(backwards) in the template. To adjust for this, the values to stop the motor and move it forwards or backwards had to be experimentally determined.

A major aspect that affected the sensor readings to determine the thresholds was the major presence and variability of sunlight reflected on the maze which was set up for the demonstration. Depending on the time of the day, the amount of sunlight present affected the readings provided by the sensors and affected the precision and consistency of the navigation of the micromouse robot.

Chapter 2

Design and Implementation of Navigation

2.1 Improvements from Milestone 2

Milestone 2 involved programming the mouse to be aware of its surroundings in some capacity, and attempt to navigate based on its surrounding awareness.

During development for Milestone 3, it was decided that the hard-coded sensor threshold approach was not robust enough to allow the micromouse to eventually solve the maze and could not be integrated with any path-finding/maze-solving algorithms needed to solve the maze. The line-following logic from Milestone 2 was therefore abandoned in favour of highly-effective yet simple programming, which borrows concepts such as error and negative feedback learnt in control systems engineering and signals & systems engineering courses in order to improve the navigation of the micromouse.

Updated Configuration of Motors

Before much of the new line-following logic could be implemented, another update to the ADC configuration for each motor. Absolute value blocks were added into the template to allow the wheels of the micromouse to spin in both the forward and reverse directions.

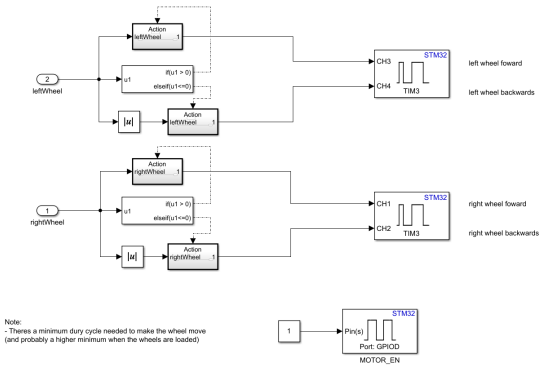

Figure 2.1 Updated motor configuration allowing for negative duty cycles to be registered

2.1.1 Using Signal Error and Negative Feedback to Implement Line-following

The Sensing Subsystem of the micromouse includes two downward-facing sensors, which are symmetrically placed about the center line of the micromouse. Due to their symmetrical placement, when the micromouse is perfectly centred on a black line, both sensor readings will have the exact same voltage output. If the micromouse drifts to the left or right of a black line, one of the downward sensors will be closer to the black line than the other, resulting in both sensors having different values.

Using this property in tandem with a control systems approach, the difference between the two downward-facing sensors can be described as the Error of the line-following signal, illustrated by the equation below:

$$Error = Sensor_{Right} - Sensor_{Left} = DOWN_{RS} - DOWN_{LS}$$

When a sensor is above a black line, most of the IR light is absorbed by the black line, resulting in a low voltage output from the sensor. This output voltage increases dramatically when sensors are above reflective objects which are not also black. By this logic, if the rightmost sensor is higher than the leftmost sensor, it indicates that the micromouse is too far to the right, and vice versa. Therefore, using the equation above, the Error signal will become positive if the micromouse is too far to the right of a black line, and negative if it is too far to the left of the black line.

Case: $Error = 0$

When the Error is equal to or, for the purposes of this project, within a certain range about 0, it indicates that the micromouse is successfully tracking the black line. In this state, both wheels will be set to the same forward speed to follow the line. The middle onboard LED will be set to flash when in this state.

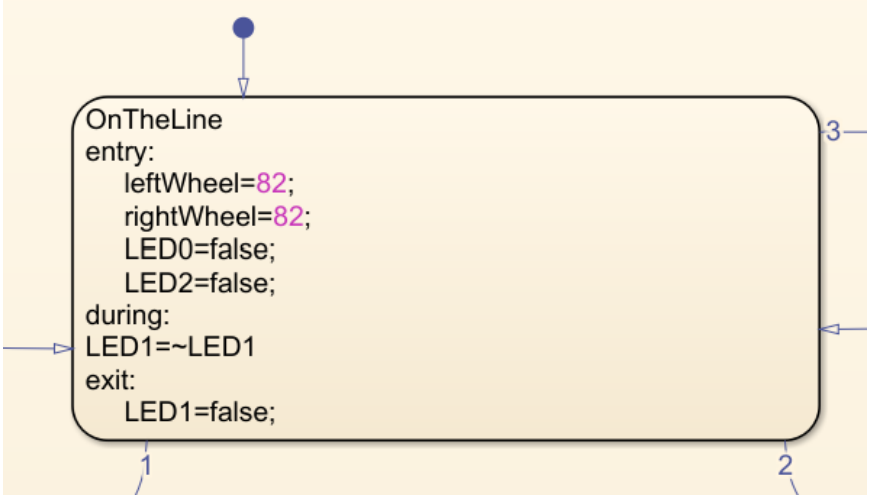

Figure 2.2 'OnTheLine' State when following a line

Case: $Error > 0$ **or** $Error < 0$

As discussed above, a positive Error signal indicates a position too far to the right of a black line. Therefore, when the micromouse is in this state, the right onboard LED will be turned on, and the left wheel of the micromouse will be gradually slowed down by a speed of 1 before in reaches its minimum forward speed. Doing so will allow the micromouse to turn towards the left, while still moving forward along the line. The same logic will be applied to the right wheel when the Error signal becomes negative. The right-side states denoting the above logic can be seen below.

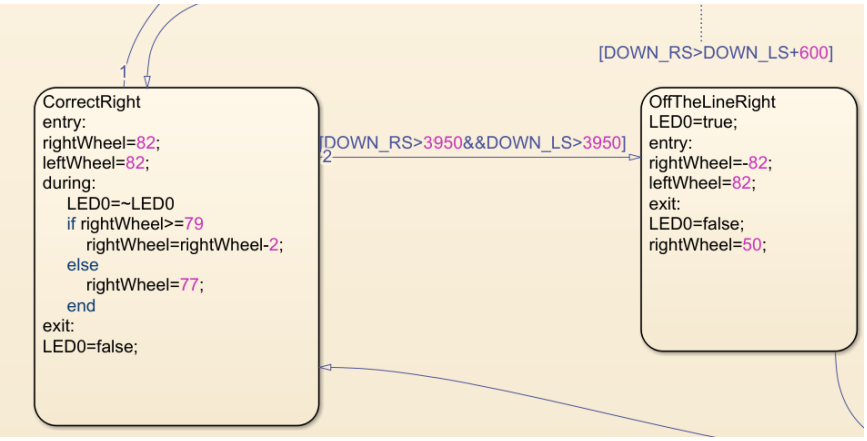

Figure 2.3 Movement correction when attempting to correct the micromouse to the right

Case: Both sensors are off the black line

When this case occurs, both sensors will read the same high output voltage, resulting in the difference between them being 0. This could be confused for false line detection, from which the micromouse will not recover from.

In one case, both sensors output low voltages, and the difference between these are 0. In the other case, both sensors output high voltages, and the difference between these are also 0. By simply checking if both sensor voltages are extremely high while the error is still zero, the micromouse will be programmed to rotate on its own axis until at least one sensor finds the line again. This rotation will be done by setting one wheel to a positive speed (forward-spinning), and the other to a negative speed (reverse-spinning). In this state, all onboard LED's will be turned on until the micromouse detects a line again. This state can be seen in Figure 2.3.

2.1.2 New Navigation Logic Based on Updated Line-following

Since this method of line-following is only dependent on the relative difference between two signals, it is highly dynamic, requiring no calibration and functioning correctly in any ambient condition, which is an extremely massive improvement from Milestone 2.

All the above states have been grouped together to form the 'LineFollowing' superstate, pictured below.

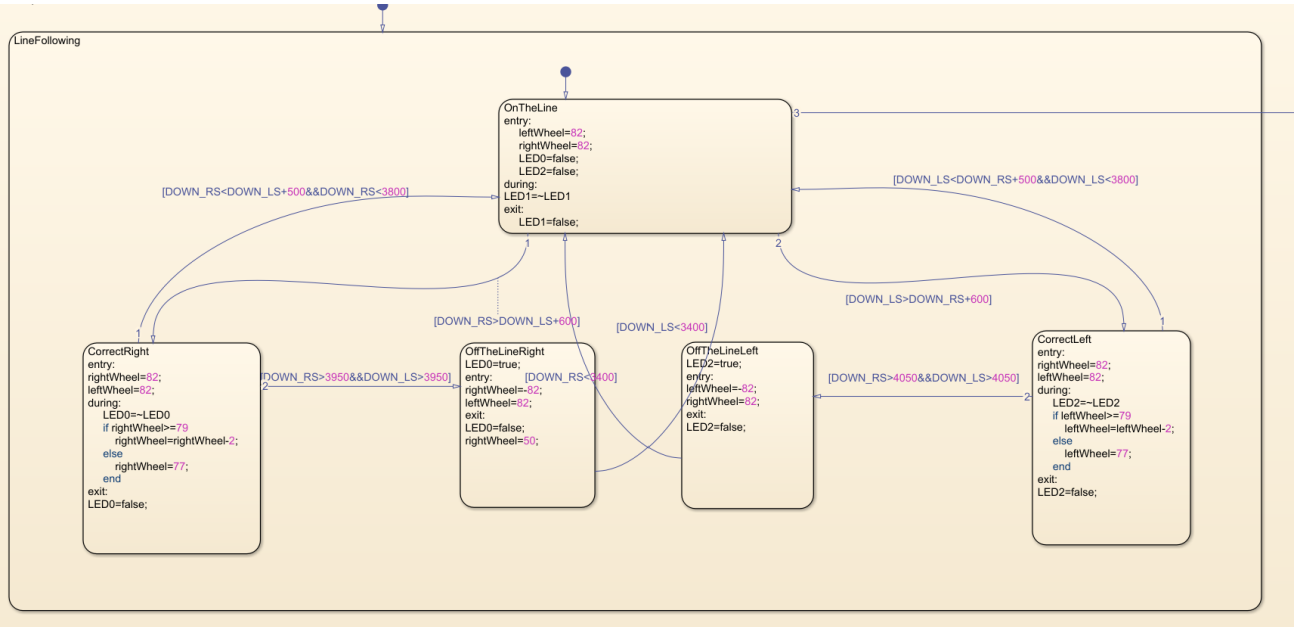

Figure 2.4 MATLAB Stateflow Logic for Line-following behaviour

As seen above, the transition between states is primarily dependent on the relationship between the downward sensors, DOWN_RS and DOWN_LS. Offset values have been added to allow the the Error signal to have a slight range of acceptable values before it starts correcting, in order to allow the micromouse to actually move forward, since it will never perfectly be equal to 0 constantly.

2.1.3 Intersection Detection using the Motor Sensors

Both motor-sensing sensors were bent 90° downwards in order to sense the black lines on the far sides of the mouse at each intersection, as they were not being used for anything else in this approach to programming the micromouse.

Initially, ideas from signals & systems engineering would have been used to obtain the continuous derivative of each motor sensor. When the derivative of these signals pulsed in the negative direction, it indicated an intersection, and produced extremely robust results. Unfortunately due to the inherent design of the Simulink Template provided, continuous transformations of any signal could not be done. And when attempting discrete derivatives or integrals, the value of each sensor would stay static, instead of being continuously polled as usual.

To account for this, the value of each motor sensor was stored every 0.05 seconds (the sampling time of all components within the micromouse). While in the 'LineFollowing' state, the previous motor sensor values would be subtracted from the respective current values. If this value became negative, it indicated an intersection, and transitioned the program to the 'IntersectionSensed' state. This implementation was extremely responsive, which is why the noisy side-facing sensors in the Sensing Subsystem were not used, as their readings were not as reliable as this method of intersection detection.

Once intersections were detected, the micromouse paused for 1 second, and the optimizing algorithm used for the micromouse dictated in which direction it would turn. In order to achieve 90° turns to the left and right, the corresponding motors were set to speeds of ±82 for exactly 0.9 seconds. This was rigorously tested, and a speed of 82 of 0.9 seconds turned the micromouse very close to 90° perfectly. Any error made when turning was accounted for by the extreme robustness of the new line-following implementation, meaning the micromouse automatically corrects itself back onto the line if a turn was just a few degrees off. This behaviour can be seen in the 'IntersectionSensed' state in Stateflow. The decision making at turns will be discussed further in chapter 3.

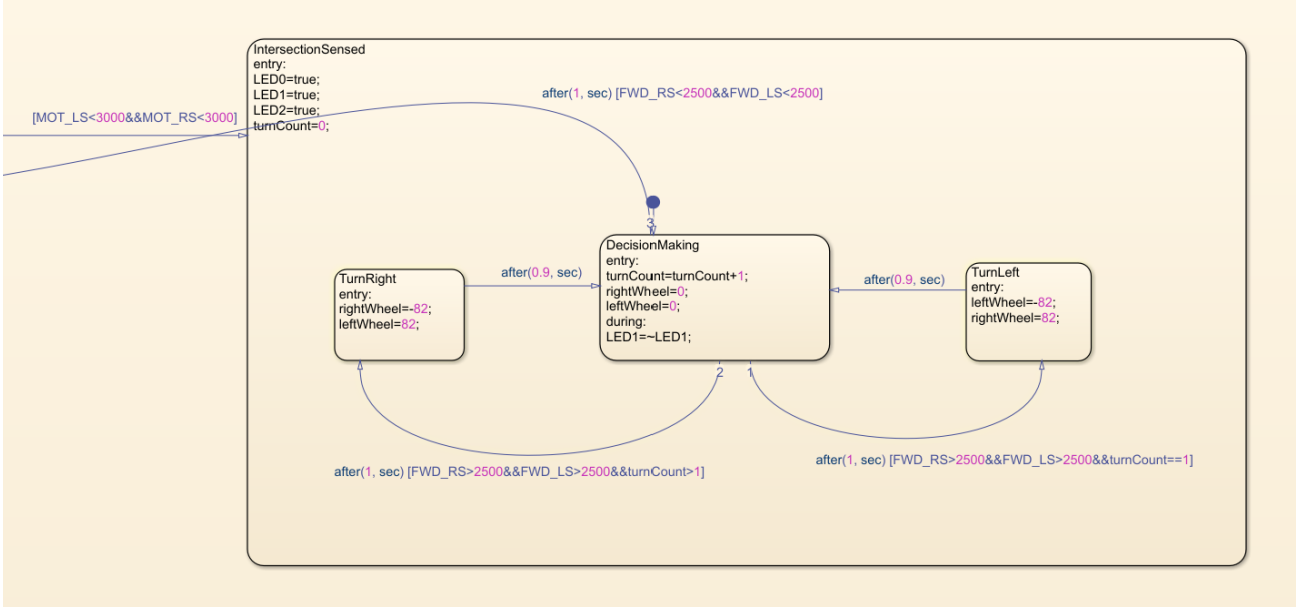

Figure 2.5 MATLAB Stateflow Logic for Sensing Intersections

Chapter 3

Design and Implementation of Optimization

3.1 Maze-solving: Following the left-hand wall

As with most implementations in this project, the maze-solving algorithm described here and implemented in the code, as well as the optimization algorithm that was intended to be used could not be performed in the demo or by either micromouse. This was not due to any error in programming, but by the duty cycle of specifically the right motor of the micromouse. When attempting to change this value from a positive speed to a negative speed within the same runtime, the right motor halts and refused to turn any more after that point. If the program initially sets the wheel speed to negative, the motor spins in reverse as intended, which indicates that it was not a hardware problem, and was an issue that ultimately prevented the micromouse from solving the maze. It should be noted that this issue was not present in the left motor, for which the code and logic was exactly the same as that of the right motor. A solution for this issue could not be found, even after approaching tutors and external guides for support.

Despite this obstacle, the report will still discuss the intended design and maze-solving algorithm.

3.1.1 Decision-making at Intersections

While the intended method of optimization would have been the implementation of a Flood Fill algorithm, due to time constraints and debugging the technical difficulties described above, the micromouse was programmed to solve the maze by following the left wall of the entire maze, also known as the Wall Follower algorithm.

The left wall following method works best in mazes with connected walls (also known as 'simply-connected mazes' or perfect mazes). These mazes have no loops and only one path from start to finish. If a maze is connected to the outer boundary, following the left-hand wall guarantees you will eventually reach the exit or return to the start after exploring all options. While this is not the most efficient algorithm, it proved to be the most effective to implement given all other difficulties, since the maze that needs to be solved fits all criterion to be a simply-connected maze.

This algorithm was implemented by programming the decision-making of the micromouse at each intersection. When stopped at an intersection:

- If there is no wall detected by the forward facing sensors, the micromouse will transition back to the 'LineFollowing' state and continue to move forward.
- If a wall is present in front of the micromouse, it will attempt to turn left instead.
- If another wall is present after turning left, the micromouse will make two right turns and attempt to turn right.
- If a third wall is also present (i.e. the micromouse has reached a dead end), it will make one further turn and move in the opposite direction from which it arrived at the intersection.

The above implementation prioritizes turning left at each intersection, making use of a local variable 'turnCount' to track how many turns have been made, and ensuring that the left wall can always be followed, and the centre of the maze will eventually be found. This implementation can be seen in Figure 2.5 in chapter 2.

3.2 Intended Flood Fill Algorithm

3.2.1 What is Flood Fill?

Flood Fill is a popular optimization algorithm for filling a certain area of 'cells,' where one cell or block can be defined as a smaller subdivision of a larger area. In the case of the 7x7 maze, a 2D array of 7 rows and 7 columns could be used to store information for each cell/block of the maze. [1]

On initialization, the cell in which traversal begins is called the 'seed,' and is assigned a cost of 0. The cost of each other cell is determined by how many cells/blocks it is from the seed. For example, cells that neighbour the seed directly would have a cost of 1, and so on. These costs are then stored in the 2D array representing the maze, and on a second optimized run, only cells with the smallest cost (i.e. the shortest path from the seed) will be traversed.

An example of how a fully processed 5x5 maze would be represented by a Flood Fill 2D array is shown below:

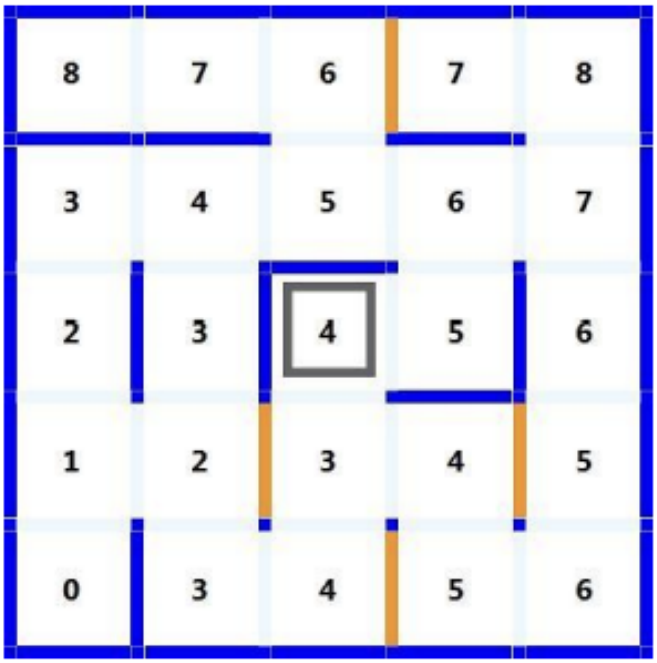

Figure 3.1 5x5 maze after all cells have been processed [1]

3.2.2 How was this intended to be implemented with the micromouse?

As discussed above, a 2D array could store the costs of each cell. The micromouse always begins traversal in the leftmost corner of the maze. During initialization, the seed could be set in the corresponding corner of the 2D array.

At first, no other cells have been processed, so by pressing SW2, the micromouse would enter its 'MazeSolving' superstate, in which it would use the left wall following algorithm described above to traverse most of the maze, process the necessary cells, and update all costs of processed cells.

Using these updated costs, the 'OptimizedSolving' superstate could be accessed by pressing SW1, in which the micromouse could perform a second, more optimized run of the maze. The micromouse would read the costs stored in its 2D array, choosing to traverse only the cells with the least cost (i.e. the shortest path), using the Flood Fill algorithm to dictate its decision-making at each intersection.

This was the intended method of optimization, but due to time constraints, technical difficulties, and administration changes, this could not be implemented successfully.

Chapter 4

Conclusion

4.1 Review of Project Achievements

In conclusion, the micromouse project achieved substantial progress across all milestones, from hardware assembly design and testing to the implementation of navigation and the maze solving logic. The use of Simulink and Stateflow models provided a robust interfacing framework to program the micromouse and to debug the logic for sensor calibration and motor control. Despite the limitation with the operation of the right motor as described in chapter 3 the micromouse successfully was able to follow a straight path and dynamically correct its line following in case it shifted away from the line. The micromouse was also successfully able to detect every intersection indicated by the tape crossing by stopping at them and was therefore able to successfully stop before reaching a wall.

However the turning logic implemented using the left wall following method only worked on the left wheel, and was not able to turn the robot due to the issues described with the right motor. As a result the maze solving algorithms could not be implemented but the design is discussed in detail in chapter 3 and should work successfully.

Overall the project highlights the importance of integrating hardware and software components to implement real-world problem solving using robots.

4.2 Future Improvements

The issue with right motor control needs to be debugged in depth by investigating the motor signals or upgrading the motor controller hardware for smooth navigation.

Integration of the gyroscope in the navigation logic for angle tracking and implementing the turning of the robot can result in more accurate turning and orientation, this can also help enhance overall movement precision.

The placement of sensors as discussed in chapter 1 in the limitations, should be corrected to be able to utilise the left and right sensors which would be able to detect the left and right walls of the maze accurately, which can be used to implement the navigation. The placement of the maze in consistent environmental conditions, unaffected by sunlight, can help in ensuring more stable navigation.

Bibliography

[1] S. Tjiharjadi, M. C. Wijaya, and E. Setiawan, "Optimization maze robot using a* and flood fill algorithm," *International Journal of Mechanical Engineering and Robotics Research*, vol. 6, no. 5, pp. 366–372, 2017.