Final Design Report

Prepared by:

Group 21

Imaan Shaik – SHKIMA004

Safiya Mia – MXXSAF002

Prepared for:

EEE3099S

Department of Electrical Engineering

University of Cape Town

Date:

22 October 2024

Table of Contents:

Executive Summary

Introduction to the Project

This report outlines the design, development, and optimization of a micromouse robot capable of autonomously navigating and solving a maze using sensor inputs, control algorithms, and strategic optimisations. The primary goal was to build a reliable system that could efficiently traverse a maze, making real-time decisions based on environmental inputs. The project aimed to implement robust navigation mechanisms and enhance them through iterative testing and improvements across two major milestones.

What Was Required

In Milestone 3, the focus was on developing a complete autonomous navigation system that could reliably guide the micromouse through the maze. This system needed to:

- Detect walls and intersections accurately using multiple sensors.
- Make real-time path adjustments to stay on course.
- Implement an algorithm to avoid dead ends and infinite loops.
- Achieve smooth movement, including precise turns and stable corrections, while maintaining speed and accuracy.

The system also required state-based transitions and timer-regulated actions to ensure consistent performance.

Milestone 4 focused on optimizing the software and hardware components to further enhance the micromouse's speed, precision, and overall performance. This phase involved:

- Reducing interference and improving sensor accuracy with hardware modifications.
- Refining control algorithms to ensure smoother transitions and better alignment during complex manoeuvres.
- Introducing the Flood Fill algorithm to find the shortest path through the maze, though full real-time recalculation capabilities were only partially implemented due to time constraints.

What Was Done

In Milestone 3, a state-machine-based navigation system was developed. It included:

- The `moveFWD` state for continuous movement, keeping the micromouse aligned with the path using sensor inputs.
- `INT` states to handle intersections, ensuring the robot could detect and navigate T-junctions and crossroads.
- Gyroscope-guided turning states to execute precise turns based on the robot's orientation.
- A priority-right algorithm that prioritized right turns, followed by forward movement, and left turns as a last resort, solving the infinite loop issue observed in earlier testing.

With timers regulating state transitions, the micromouse was able to maintain smooth and consistent performance. This system was successfully demonstrated as the micromouse navigated to the center of the maze and returned to the starting point, meeting the core requirements of Milestone 3.

In Milestone 4, the focus shifted toward optimisation. Tape and Prestik were added to the sensors to reduce ambient light interference and enhance wall detection accuracy. Software improvements were made to the Tune state to align the micromouse more effectively after sharp turns or periods of fast movement. The introduction of the Flood Fill algorithm further optimized pathfinding, though real-time path recalculation remained incomplete due to time constraints. These optimizations resulted in smoother movements and improved performance, though some challenges, such as occasional gyro loops and sensor misreads, persisted. These insights have provided valuable direction for future improvements in subsequent phases.

Milestone 3: Design and Implementation of Navigation

1. Milestone Definition and Requirements:

Milestone 3 presented arguably the most important challenge of the entire micromouse project - to design a navigation system capable of autonomously navigating the micromouse through a maze. More specifically, this would require:

- Accurate detection of walls and intersections using multiple sensors.
- Dynamic path adjustments based on real-time sensor feedback.
- An algorithm that avoids potential infinite loops or dead ends.
- Efficient movement, including turns and course corrections, while maintaining overall speed and precision.

2. Design Approach:

The state machine helps in breaking down the maze navigation task into smaller, manageable steps. Each state has a clear responsibility (e.g., detect walls, turn left, move forward), which ensures that the micromouse makes decisions incrementally based on sensor inputs. These states, along with their transitions (defined by the overarching navigation algorithm), enable the micromouse to successfully complete the milestone. The logic behind each step of the solution is explained in the following subsections:

2.1 Forward Movement (`moveFWD`)

The micromouse relies on multiple sensors to detect walls and intersections. These include front, side, and downward-facing sensors. Before deciding on a navigation algorithm, the following states and logic were implemented:

The micromouse moves forward continuously, adjusting its path based on input from the downward sensors (`ADC13_DOWN_LS` and `ADC10_DOWN_RS`), to remain on the black line. Any discrepancy between these sensor readings prompts the micromouse to correct its course by adjusting the motor speeds. To maintain alignment, the system calculates the difference (`d`) between the readings of the two sensors. This value represents how far the micromouse has deviated from the centre. A positive or negative value of `d` indicates whether the micromouse is drifting toward the left or

right side of the black line. Using this deviation, the speeds of the wheels are dynamically adjusted to correct the course. Specifically, the left wheel's speed decreases, and the right wheel's speed increases proportionally. This ensures that the micromouse gently steers back towards the centre without abrupt movements. Additionally, safeguards are in place to cap the wheel speeds within a specific range (70 – 80) to maintain control and prevent overshooting. By adjusting the speeds incrementally based on sensor input, the micromouse avoids oscillations and ensures smoother movement. For example, if the downward sensors detect that the micromouse has drifted slightly to the left, the right wheel speed is increased slightly (`e.g., rightWheel = rightWheel + round(d / 100)`) to push the micromouse back to the centre. The correction factor of 100 was found to provide smooth course corrections by introducing sufficiently small, incremental adjustments to the wheel speed. The details of how this correction factor was determined is documented in Table 7.

2.2 Wall Detection (`wall_detect)`:

At intersections, the micromouse uses its front sensors (`ADC9_FWD_RS` and `ADC14_FWD_LS`) to detect walls. The sum of these sensor readings determines the presence of obstacles ahead. The micromouse also uses side sensors for detecting walls on the left and right, and the corresponding LEDs (`LED0`, `LED1`, and `LED2`) are activated to indicate the location of detected walls.

2.3 Intersection Handling (`INT, INT_side, INT_next`)

The micromouse follows a multi-step process to reliably detect and navigate intersections:

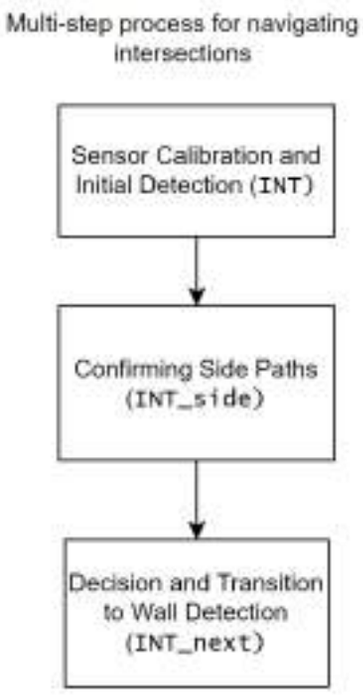


Figure 1: Multi-Step Process followed by micro-mouse during navigation

The `INT` state collects initial sensor data when the micromouse first enters a potential intersection. This state ensures that the micromouse stops or slows down at the intersection to avoid overshooting. The forward sensors are read to check if the path is clear or blocked. The micromouse also stores the previous wheel speeds to resume them after crossing the intersection.

The `INT_side` state focuses on side sensors to determine if the micromouse has valid paths to turn left or right. This step helps the micromouse confirm that there are side openings, which indicates a T-junction or cross-intersection. It might also determine whether both side paths or only one is available.

The `INT_next` state checks the final state of the side sensors before deciding the next move. The micromouse ensures that all relevant sensors are in sync (i.e., clear readings) before transitioning to wall detection or movement logic. This state could also confirm that the intersection data gathered earlier is still valid (i.e., no new obstacle appeared suddenly).

2.4. Turning and Course Correction (`turnCW, turnCCW, turnBack`)

When the micromouse detects walls or reaches intersections, it can execute turns using the following states, `turnCW`, `turnCCW`, and `turnBack`. The turning mechanism is guided by gyroscope readings to ensure precise orientation. The use of gyroscope data ensures that turns are executed with precision, which is critical in a maze environment where small errors can lead to collisions or inefficient paths.

Analysing the `turnCW` state and associated logic (the same logic can be applied to implement `turnCCW` and `turnBack`):

A right turn has been executed when the gyroscope reading (`gyroSensor`) is less than `angularOrient - 75`. The condition checks if the current gyroscope reading (`gyroSensor`) is at least 75 degrees less than the initial orientation (`angularOrient`), which ensures the micromouse completes a proper right turn. As the robot rotates clockwise, the gyroscope reading decreases, and when the angle difference reaches 75 degrees, the turn is considered complete. The –75 degree threshold was selected after testing various angles in Table 1, balancing the need for speed and precision. It ensures the micromouse makes reliable right turns without overshooting or under-rotating, even in the presence of sensor noise or wheel slippage.

Trial #	GyroSensor Threshold (Degrees)	Observations
1	-45	Under-rotated, misaligned with path.
2	-60	Better, but slight drift noticed.
3	-75	Smooth and aligned turn.
4	-90	Turn accurate but too slow.

Table 1: Various Angles Trialled in determining the Gyroscope Threshold

To actually execute a right turn, the micromouse increases the left wheel speed to 90 and reduces the right wheel speed to 63. This speed differential makes the micromouse pivot to the right. This condition ensures that the turn completes only if the right sensor detects a clear path (`detectRight < thresRight`), and the left sensor identifies an obstacle (`detectLeft > thresLeft`). This guarantees that the robot makes the right turn correctly.

Trial #	Left Wheel Speed	Right Wheel Speed	Turn Quality
1	80	60	Turn too wide, overshot the path.
2	80	65	Better control, slight overshoot.
3	80	63	Smooth, precise turn with no overshoot.
4	80	60	Turn too sharp, caused skidding.
5	80	62	Nearly optimal, but slightly slow.

Table 2: Table showing how the reduced right wheel speed was determined

Course corrections during forward movement are controlled through the **`moveFWD`** state and detailed in the previous section.

2.3. The Navigation Algorithm

The navigation algorithm follows a series of rules for directional decisions. Initially, a forward-priority algorithm was considered, where the micromouse would prioritise moving forward. However, this approach introduced a significant problem, which is described in further detail:

It was observed that when the micromouse reached a dead end whose wall is directly opposite the wall of another dead end, it would reverse and attempt to move forward again, encountering the same dead end repeatedly.

This is visually represented in the figure below:

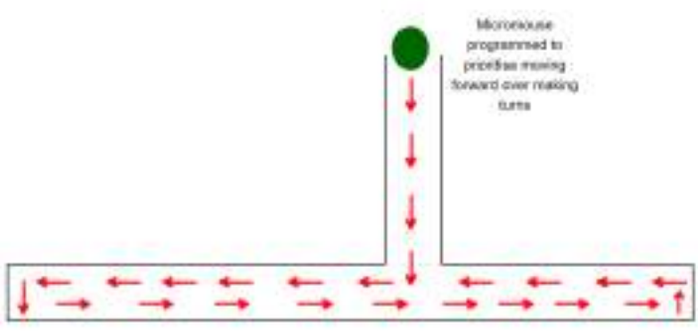


Figure 2: Figure showing the problem with a forward-priority algorithm

Thus, this problematic algorithm caused an infinitely looping trajectory for the micromouse in any maze segment configured like the one in Figure 2.

To address this, the **priority-right algorithm** was introduced:

- **Right Turn First:** When the micromouse encounters a junction or an open path, it checks for a right turn first. If a right turn is available, the micromouse takes it.
- **Forward Movement Second:** If no right turn is possible, the micromouse tries to move forward. It only proceeds straight if the path is clear.
- **Left Turn Last:** If both the right and forward paths are blocked, the micromouse finally turns left. This ensures it doesn't backtrack unless absolutely necessary.

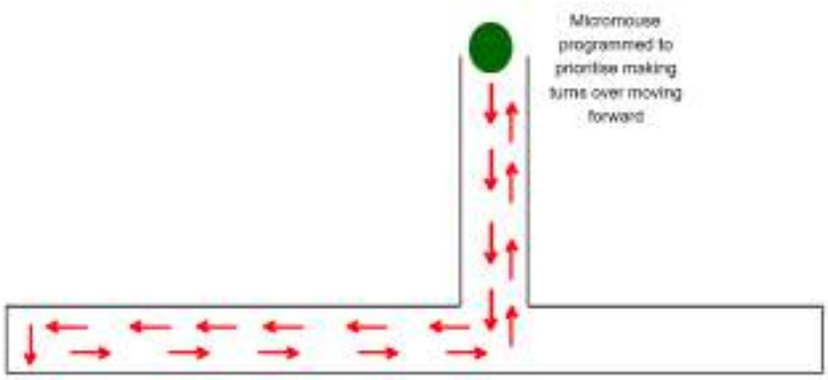


Figure 3: Figure showing how a turning-priority algorithm solves the problem created by using a forward-priority algorithm

As observed in Figure 3, by prioritising turns over forward movement, this approach eliminates the infinite loop problem and ensures more efficient exploration of the maze.

2.4. Timing and State Transitions

Transitions between states are regulated by timers to ensure smooth execution of actions:

- For example, after(0.25, sec) ensures that an action (e.g., turning) is performed for 0.25 seconds.
- Similarly, after(0.45, sec) adjusts the wheel speeds after 0.4 seconds to prevent overcorrection during manoeuvres.

This timing mechanism allows the micromouse to maintain consistent performance, avoiding abrupt movements or excessive corrections during turns or path adjustments.

Trial #	Timer Value for Turns	Timer Value for Path Adjustments	Observations
1	0.1	0.25	Turn incomplete; overcorrection occurred.
2	0.15	0.3	Improved, but small overshoot during turns.
3	0.25	0.45	Smooth turns, stable alignment after turns.
4	0.3	0.5	Sluggish response, slower course corrections.
5	0.2	0.35	Nearly optimal, but slightly faster than ideal.

Table 3: How the Timing Intervals Were Found

With 0.2 seconds for turns and 0.4 seconds for adjustments, the micromouse achieved optimal smoothness and accuracy. The slightly longer adjustment period (0.4 sec) ensured that corrections were applied gently, avoiding overcorrection and abrupt movements.

3. Results:

The micromouse successfully navigated the maze using sensor-based state transitions and the priority-right algorithm, avoiding dead ends and completing the maze.

Metric	Value	Interpretation

Completion Rate	80%	8 out of 10 runs reached the centre.
Average Completion Time	62s	On average, it took 112 seconds to complete.
False Positive Rate	10%	2 incorrect wall detections in 20 attempts.
Missed Wall Rate	20%	Missed 3 walls out of 15 in total.
Correct Turn Rate	93.33%	28 out of 30 turns were correct.
Average Time Per Turn	1.5s	Each turn took 1.5 seconds on average.

Table 4: Performance evaluation metrics from the final Milestone 3 tests.

Based on the test results recorded in Table 4, the micromouse recorded a false positive rate of 10% and a missed wall rate of 20%. Several factors likely contributed to these issues. False positives (incorrect wall detection) can arise from sensor noise, reflections, or miscalibration. For instance, reflective surfaces or uneven lighting conditions within the maze may cause sensors to incorrectly detect obstacles where none exist. This can lead to unnecessary turns, slowing down the micromouse and increasing its navigation time. On the other hand, missed walls (where the micromouse fails to detect a wall) may be due to sensor blind spots, limited range, or environmental interference. If the walls were positioned just outside the sensor's effective range or the sensor readings were inconsistent, the micromouse might proceed without realising a wall is present. This was evident in some runs, where missed detections caused the micromouse to bypass critical turns, contributing to incorrect paths and redundant movements. These issues also impacted the completion rate (80%), as missed walls and unnecessary turns delayed the mouse's progress through the maze. These detection challenges will be eliminated in Milestone 4 by improving sensor alignment, increasing sensitivity, and reducing environmental noise to significantly enhance the micromouse's navigation accuracy and overall performance.

Milestone 4: Design and Implementation of Optimisation

1. Milestone Definition and Requirements:

The optimisation phase of the micromouse project was centred on improving performance, speed, and navigation accuracy. After the basic navigation system was in place, multiple optimisations were necessary to ensure the micromouse could efficiently navigate the maze and reduce the completion time.
Optimisations were implemented in both the software and hardware components of the micromouse to improve the overall efficiency.

The key focus areas included refining certain algorithms, optimising sensor performance, and improving the speed and accuracy of the maze traversal.

2. Design Approach:

2.1. Hardware Optimisations

The following modifications to the hardware were made:

- Applying tape over the left and right sensor LEDs:

Each of the sensors were highly sensitive to fluctuations in ambient lighting throughout the course of the day. Black tape was wrapped around the sensors in a tubular shape, forming an encasing known as a “hat”. This was implemented to minimize the crosstalk between the forward-left and forward-right infrared sensors. This modification effectively mitigated interference from overlapping fields of view, which had previously impeded wall identification and resulted in inaccurate readings.

Following this implementation, the sensitivity stabilised by a significant amount and the outcome of this modification is tabularized in Table 6.

Front Facing Sensors		
	Right	Left
min		
	1.31	0.95
	1.28	0.97
	1.27	0.91
max		
	1.41	1.16
	1.45	1.1
	2.42	2.11
mean		
nothing	1.41	1.23
blocked	3.3	3.3
turning pnt	1.45	1.1

	left mean	right mean
max	1.456667	1.76
min	0.943333	1.286667

Total left mean	1.2
Total right mean	1.523333

ADC left max	1807.591
ADC left min	1170.591
Mean	1489.091

ADC right max	2184
ADC right min	1596.636
Mean	1890.318

Threshold mean	1689.705

Table 5: Sensor data collected prior to the addition of tape

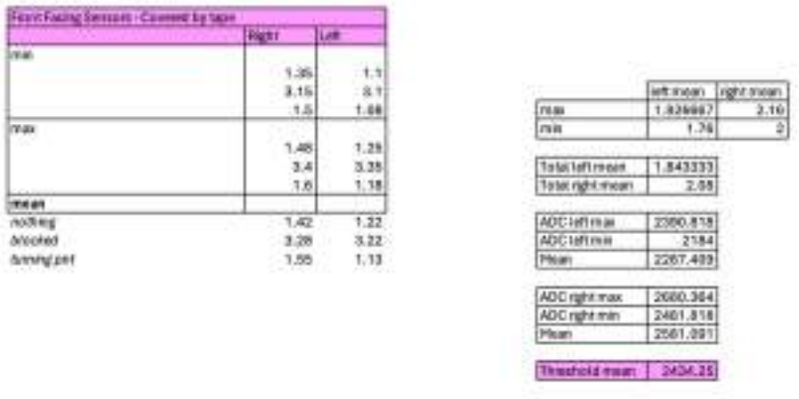


Table 6: Sensor Data collected after covering the sensors with tape

- **Covering the back of the sensor LEDs with Prestik:**

Further optimisation of sensor performance was achieved by covering the backs of the taped sensors with Prestik. It was noted that ambient light could potentially "leak" into the sensors from the rear, affecting their accuracy. By applying Prestik, the exposure to extraneous light was minimized, thereby reducing interference and enhancing the sensors' ability to detect walls and obstructions more reliably. This additional measure contributed to the overall precision and reliability of the micromouse's navigation capabilities, ensuring consistent performance in varied lighting conditions. Tests were performed on the sensor readings prior to this modification and compared to the results yielded after this implementation.

- **Soldering of the High Power Jumpers**

During the assembly of the micromouse, the Low-Power jumpers were initially soldered. Further soldering of the High-Power jumpers was implemented to enhance the brightness of the sensor LEDs relative to the surrounding ambient light. This

adjustment significantly reduced the influence of ambient light, thereby improving the accuracy and reliability of the sensor readings.

2.2. Software Optimisations

The creation and implementation of a `Tune` state:

The `Tune` state functions as a critical **fine-tuning state** that ensures the micromouse remains precisely aligned with the path, correcting any minor deviations that occur during forward movement or turns. As the micromouse navigates the maze, subtle misalignments can accumulate, especially during sharp turns or after high-speed forward movement, making real-time corrections essential. The turning states (such as `turnCW`, `turnCCW`, and `turnBack`) and the forward movement (`moveFWD`) state transition to the Tune state to allow for these precise adjustments. In the Tune state, the micromouse utilises readings from its downward-facing sensors (`ADC13 and ADC10`) to detect deviations from the centre of the path and applies small, gradual corrections by adjusting the speeds of the left and right wheels. The correction factor used in this state is smaller (`d/160`) compared to the forward state (`d/100`), which ensures that the adjustments are smooth, avoiding oscillations or overshooting the path. These finer adjustments are essential after turns, where the micromouse might drift slightly off course, or after periods of fast movement, where more aggressive corrections could have been applied. By constantly returning to the Tune state, the system ensures that the micromouse stays accurately aligned, providing a balance between stability and precision throughout the navigation process. This mechanism prevents cumulative errors, ensuring smooth transitions between turns and forward motion, and helps the micromouse maintain consistent, reliable movement as it explores the maze.

This table captures the iterative process of trial, observation, and adjustment used to fine-tune the correction factors, demonstrating how the final values were selected to ensure efficient and stable navigation.

Trial #	Correction Factor (`moveFWD`)	Correction Factor (`Tune`)	Stability Observed
1	80	200	Too aggressive, caused oscillations.

2	100	180	Improved but still some instability.
3	120	150	Smoother corrections, fewer oscillations.
4	130	160	Over-correction during turns.
5	100	160	Balanced corrections, stable and smooth.

Table 7: Fine-tuning the correction factor for d

The final configuration with **100** for the forward state and **160** for the Tune state provided the optimal balance between speed and stability, ensuring precise alignment without oscillation.

Flood Fill Algorithm Design

The Flood Fill algorithm is implemented as the final optimisation phase. This pathfinding technique ensures that the shortest and most optimal path to the centre cell of the maze was identified and followed. The logic of this technique is explained as follows:

Initialisation: The centre cell is assigned a value of 0, which represents 0m to the goal.
Mapping or "flooding" the Maze: The algorithm updates each neighbouring cell with a distance value that was one greater than the current cell's value. These numbers represent the distance from the centre goal cell. This is repeated until every possible cell in the maze is assigned a distance value.
Path finding: The micromouse navigates by moving to the neighbouring cell with the lowest distance value, ensuring that it follows the shortest path possible to reach the goal.
Real Time Adjustments: The map is meant to be continuously updated as the micromouse discovers new walls or obstacles, allowing for real time updated calculations to determine the optimal path.

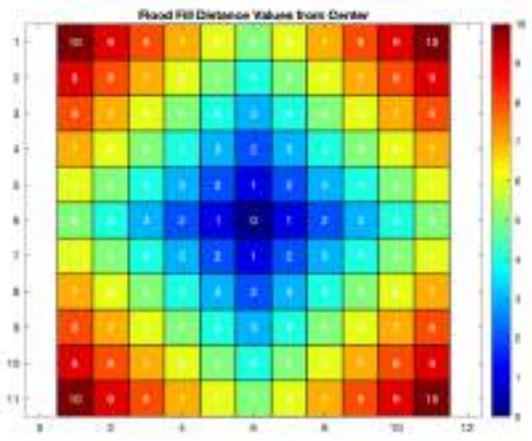


Figure 4: MATLAB generated Colour Map showcasing the formation of the Flood Fill Algorithm logic for the Maze

This logic was implemented onto the model of the actual maze to be solved and can be seen as follows:

Figure 5: Flood Fill logic mapped onto model of the maze

Implementation of the Flood Fill Algorithm:

The plan for implementing the Flood Fill algorithm included dynamic real-time path recalculations to optimize micromouse navigation based on newly discovered walls. The approach involved leveraging the use of matrix representation to simulate the maze, as well as tracing the path from the end goal to the start position. The algorithm was designed to identify viable paths by checking for available directions (forward, back, left, right) at each step and marking the traced path in a separate matrix for reference.

Using the algorithm, the micromouse was expected to dynamically adapt its movements based on sensor inputs, recalculating distances as walls were encountered. The ‘normalization’ function would simplify the maze into binary data (representing walls and open spaces), and the ‘flood’ function would propagate through the maze, marking paths while backtracking to handle dead-ends and optimizing the route.

While the Flood Fill algorithm was partially implemented to guide the micromouse to the maze's center by calculating the shortest path based on distance values assigned to each cell, time constraints prevented the completion of full integration for real-time pathfinding adjustments. Specifically, the logic for dynamically updating the distance matrix in response to sensor data remained unfinished, which limited the micromouse's ability to continuously refine its path based on newly discovered walls.

3. Results:

During the final demonstration, the micromouse successfully navigated through the maze, showcasing the line-following aspect of navigation from Milestone 3.
The `Tune` state contributed to stable movement, and the micromouse avoided collisions, validating the optimisations made to the navigation logic. However, the Flood Fill algorithm’s incomplete implementation meant that the micromouse did not find the optimal, shortest path to the centre. Instead, it traversed the entire maze, including paths that could have been avoided, resulting in a longer completion time. Once the micromouse reached the centre block, it did not stop as expected. Instead, it continued to move past the centre, driven by the priority-right algorithm, ultimately exiting the maze boundaries. This occurred because the logic responsible for halting the micromouse after reaching the centre was not fully developed.

Another issue observed during the final test was related to the micromouse’s gyroscope implementation. At several points, the micromouse became stuck in a loop, continuously turning without progressing forward. This indicated a flaw in the gyroscope’s ability to correctly sense orientation changes, leading to improper turning sequences. This behavior further contributed to the suboptimal path finding and increased traversal time.

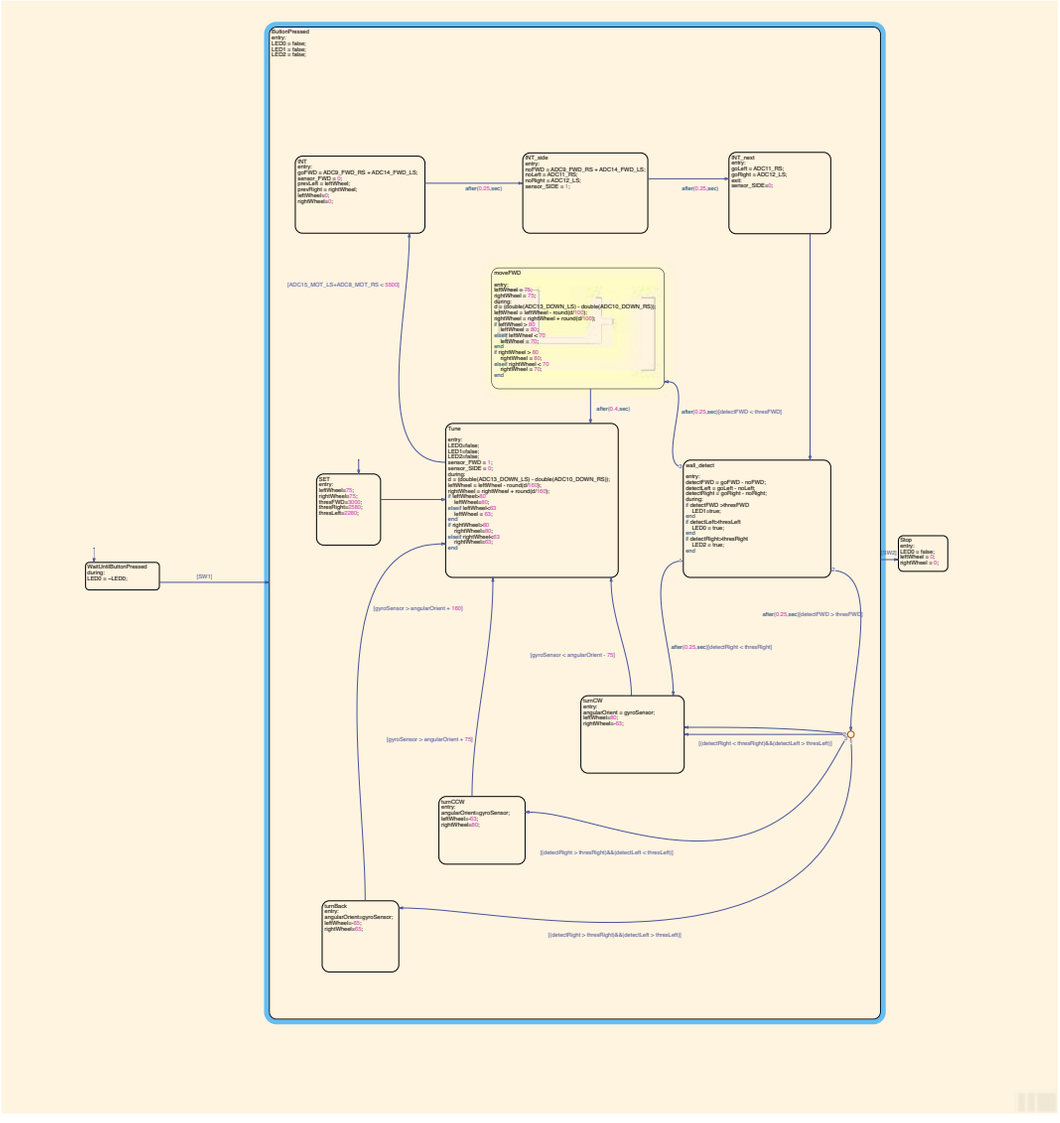
Conclusion:

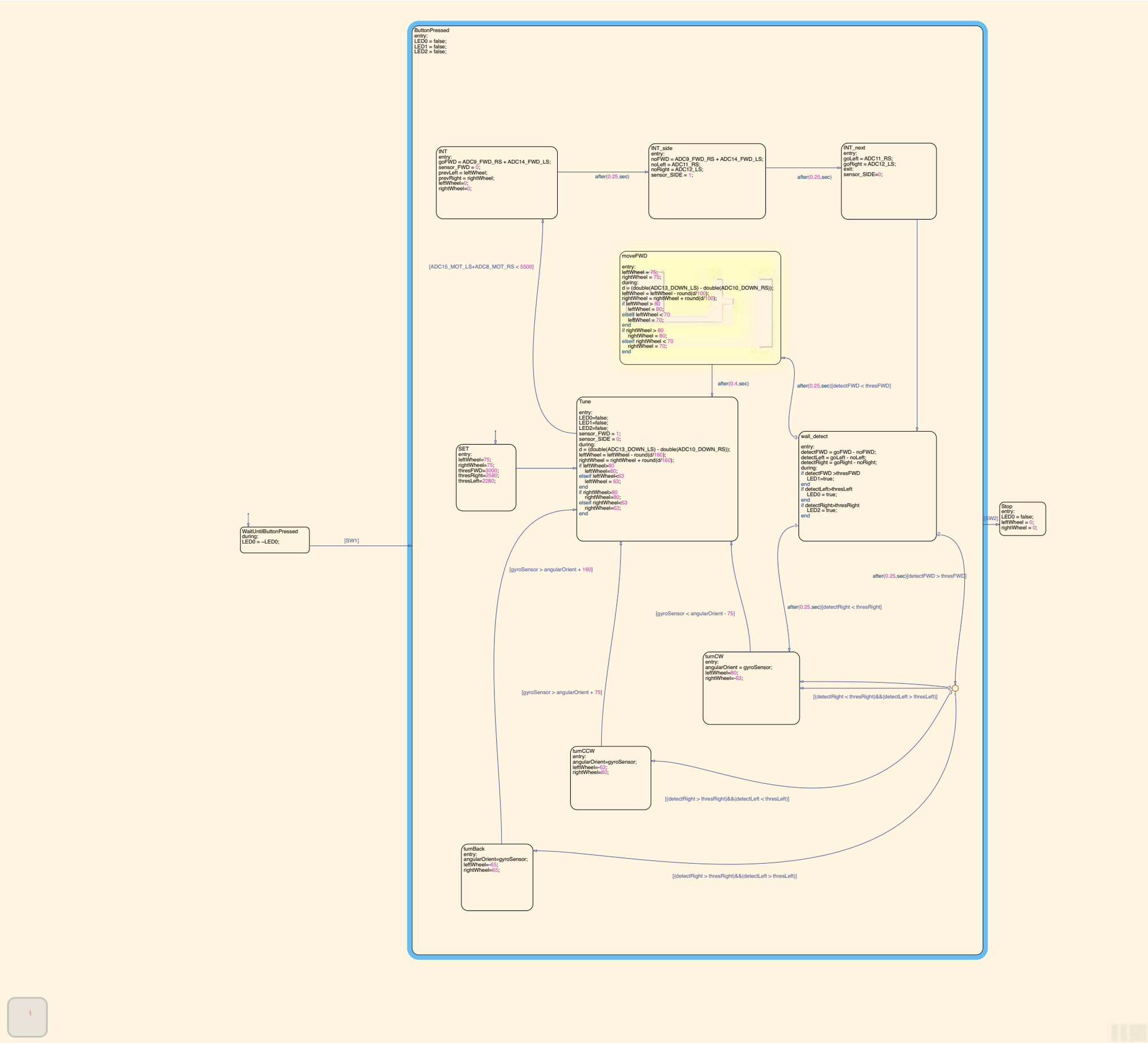
Milestone 3 successfully implemented a navigation system that enabled the micromouse to traverse the maze autonomously, validating the effectiveness of the priority-right algorithm and state-machine architecture. The navigation system designed in Milestone 3 enabled the micromouse to identify intersections, avoid dead ends, and execute precise turns using real-time sensor feedback. The priority-right algorithm ensured efficient exploration, avoiding infinite loops and minimising unnecessary backtracking. The `Tune` state effectively managed fine-tuning during transitions, reducing oscillations and improving stability. Building on this foundation, Milestone 4 introduced critical optimisations, such as sensor shielding and improvements to wheel speed calibration, enhancing the micromouse's ability to handle varied conditions. The partial implementation of the Flood Fill algorithm showcased the potential for further path optimisation, though the micromouse could not yet adapt dynamically to newly discovered obstacles.

While the micromouse successfully navigated to the maze's centre and returned to the start, some challenges persisted, including gyroscope misalignments causing the robot to get stuck in turning loops and false positive wall detections slowing progress. These insights will guide future improvements to the micromouse's logic and sensor systems. Overall, the combined achievements of Milestones 3 and 4 reflect the micromouse's evolving capability to navigate complex mazes autonomously, with clear directions for further enhancement in upcoming phases.

Appendix

Stateflow Code:




ButtonPressed
entry:
LED0 = false;
LED1 = false;
LED2 = false;
INT
entry:
goFWD = ADC9_FWD_RS + ADC14_FWD_LS;
sensor_FWD = 0;
prevLeft = leftWheel;
prevRight = rightWheel;
leftWheel=0;
rightWheel=0;
after(0.25,sec)
INT_side
entry:
noFWD = ADC9_FWD_RS + ADC14_FWD_LS;
noLeft = ADC11_RS;
noRight = ADC12_LS;
sensor_SIDE = 1;
after(0.25,sec)
INT_next
entry:
goLeft = ADC11_RS;
goRight = ADC12_LS;
exit:
sensor_SIDE=0;
[ADC15_MOT_LS+ADC8_MOT_RS < 5500]
moveFWD
entry:
leftWheel = 75;
rightWheel = 75;
during:
d = (double(ADC13_DOWN_LS) - double(ADC10_DOWN_RS));
leftWheel = leftWheel - round(d/100);
rightWheel = rightWheel + round(d/100);
if leftWheel > 80
leftWheel = 80;
elseif leftWheel <70
leftWheel = 70;
end
if rightWheel > 80
rightWheel = 80;
elseif rightWheel < 70
rightWheel = 70;
end
after(0.4,sec)
after(0.25,sec)[detectFWD < thresFWD]
Tune
entry:
LED0=false;
LED1=false;
LED2=false;
sensor_FWD = 1;
sensor_SIDE = 0;
during:
d = (double(ADC13_DOWN_LS) - double(ADC10_DOWN_RS));
leftWheel = leftWheel - round(d/160);
rightWheel = rightWheel + round(d/160);
if leftWheel>80
leftWheel=80;
elseif leftWheel<63
leftWheel = 63;
end
if rightWheel>80
rightWheel=80;
elseif rightWheel<63
rightWheel=63;
end
SET
entry:
leftWheel=75;
rightWheel=75;
thresFWD=3000;
thresRight=2580;
thresLeft=2280;
wall_detect
entry:
detectFWD = goFWD - noFWD;
detectLeft = goLeft - noLeft;
detectRight = goRight - noRight;
during:
if detectFWD >thresFWD
LED1=true;
end
if detectLeft>thresLeft
LED0 = true;
end
if detectRight>thresRight
LED2 = true;
end
Stop
entry:
LED0 = false;
leftWheel = 0;
rightWheel = 0;
[SW2]
WaitUntilButtonPressed
during:
LED0 = ~LED0;
[SW1]
[gyroSensor > angularOrient + 160]
after(0.25,sec)[detectFWD > thresFWD]
after(0.25,sec)[detectRight < thresRight]
[gyroSensor < angularOrient - 75]
turnCW
entry:
angularOrient = gyroSensor;
leftWheel=80;
rightWheel=-63;
[gyroSensor > angularOrient + 75]
[(detectRight < thresRight)&&(detectLeft > thresLeft)]
turnCCW
entry:
angularOrient=gyroSensor;
leftWheel=-63;
rightWheel=80;
[(detectRight > thresRight)&&(detectLeft < thresLeft)]
turnBack
entry:
angularOrient=gyroSensor;
leftWheel=-65;
rightWheel=65;
[(detectRight > thresRight)&&(detectLeft > thresLeft)]