Indian Institute of Technology, Gandhinagar

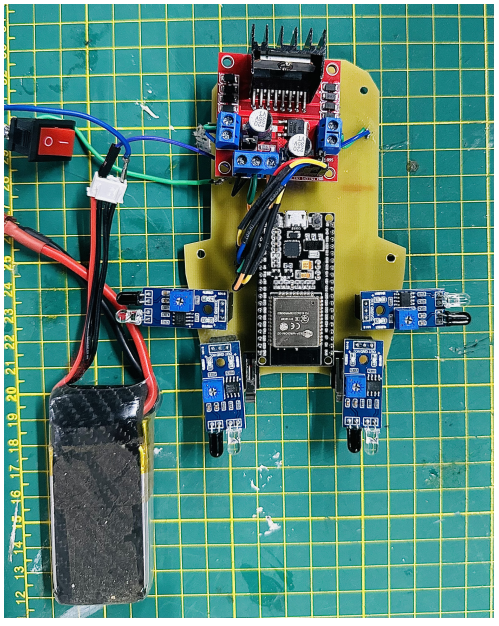
Micromouse: Team Beta

An Autonomous Maze Solving Robot

Project Course at Maker Bhawan

Authors
Hitesh Kumar
Shaurykumar Patel

Roll Number:
22110098
22110241



June 3, 2024

Contents

1 Introduction

The "Amazing Micro-mouse challenge" is a robotic competition that has been introduced in 1977. Since then, it has become a well-known introductory exercise in the field of robotics, as it is challenging enough to be non-trivial, yet simple enough to be approachable by someone without significant experience.

The Micromouse competition consists of building a robot, which can solve a previously unknown maze. First, the robot must explore the maze of 8×8 cells. It starts from a fixed corner position and must find the center of the maze, composed of four empty unit squares. After that, the robot races from start to finish, trying to succeed in the least time possible.

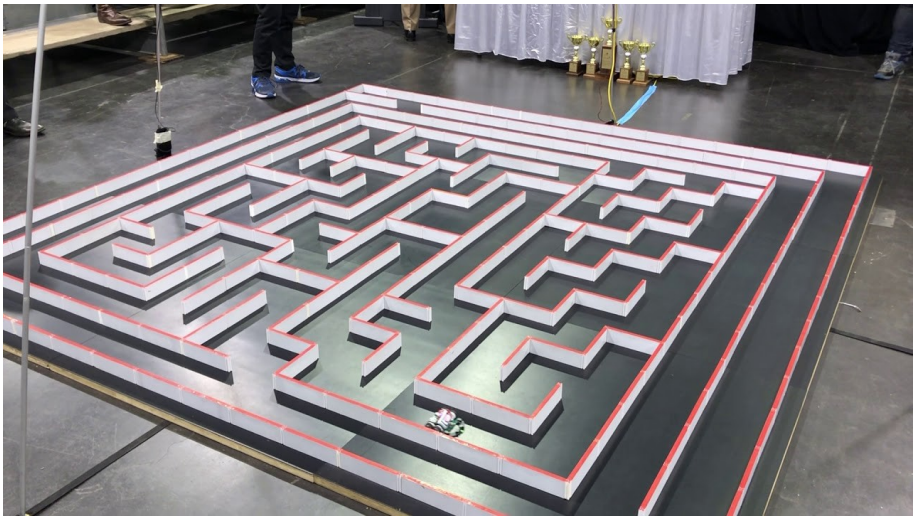


Figure 1: Example of a real-world micromouse challenge

This challenge may, at first glance, sound simple, yet entails a lot of different tasks. Starting from scratch, we had to plan our own hardware, design schematics with the help of DipTrace and construct a PCB design . Then we had to order the respective parts and solder them together.

In this report, we will describe our solution to this challenge, developed during the "Micromouse: Maze Solving Robot" Project Course offered at the Indian Institute of Technology, Gandhinagar in the Semester II 2024-2025

2 Conceptual Design

This section provides a broad outline of our approach. We include an image of our Micromouse in Figure 2 to enhance the comprehension of the subsequent sections. The dimensions of the robot are 150 mm in length, 100 mm in width, and 50 mm in height.

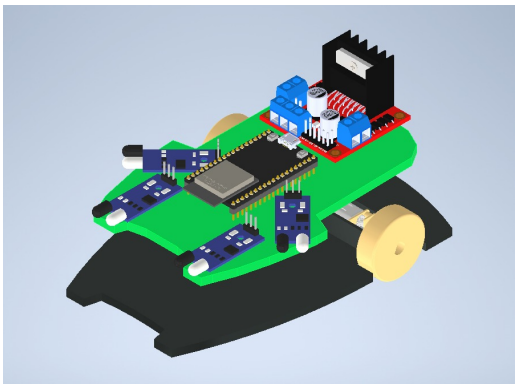


Figure 2: CAD model

2.1 List of Materials

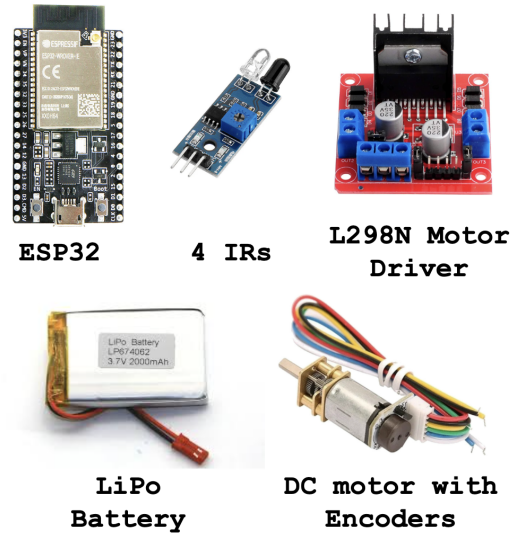


Figure 3: Materials

2.2 Hardware Overview

In addition to the rules outlined in section 1, we had several additional hardware constraints:

- The motors to be used were predetermined. We were required to use DC N20 gear motors with built-in encoders.
- The microcontroller selection was predetermined. We opted to utilize the ESP32 microcontroller for our project.

Given these constraints, we outlined a high-level overview of the hardware necessary for our Micromouse, as depicted in Figure 3.

- **Sensing:** We opted to incorporate four infrared (IR) sensors - specifically, left FrontIR, right FrontIR, Left IR, and Right IR - to detect the presence of maze walls. These sensors interface directly with the ESP32 microcontroller.
- **Acting:** The predetermined motors were integrated into our design, and we chose to drive them using an L298N motor driver module. The motor control is facilitated by the ESP32 microcontroller's PWM (Pulse Width Modulation) capabilities.
- **Support:** All the components needed to make the others work: power supply, programming headers, LEDs and switches (useful for debugging).

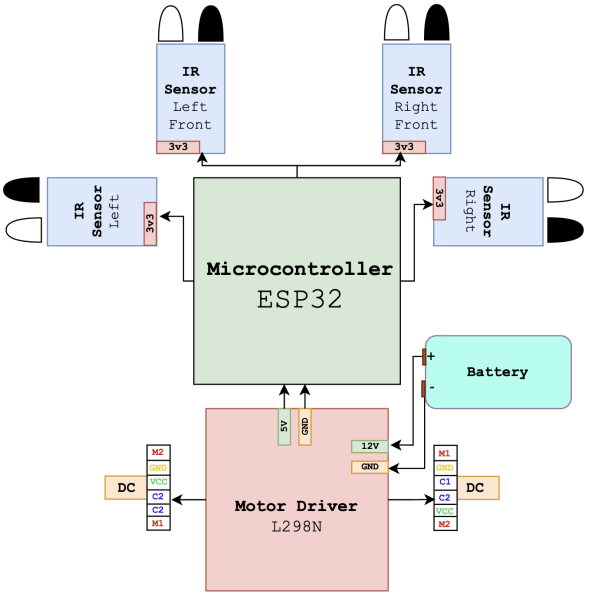


Figure 4: Block Diagram

3 Prototype on Breadboard

Testing the hardware components and assembling various testing prototypes were crucial steps to ensure the functionality of the ordered components by the teams.

For our initial prototype, we conducted tests on four Infrared Sensors (IRS) and two DC motors, utilizing the ESP32 microcontroller along with the Motor Driver L298N. This testing phase involved setting up the components on a breadboard and running basic code to evaluate their performance and compatibility.

The purpose of this prototype was twofold: to verify the functionality of the individual components and to assess their interaction when integrated into a cohesive system. By observing how the sensors responded to different environmental conditions and how the motors behaved under various control signals, we gained insights into potential challenges and optimizations required for the final design.

Additionally, this testing phase provided valuable feedback on the compatibility and efficiency of the chosen microcontroller and motor driver combination. Any discrepancies or issues encountered during testing were addressed and iteratively improved upon in subsequent prototypes, ensuring a robust and reliable final implementation.

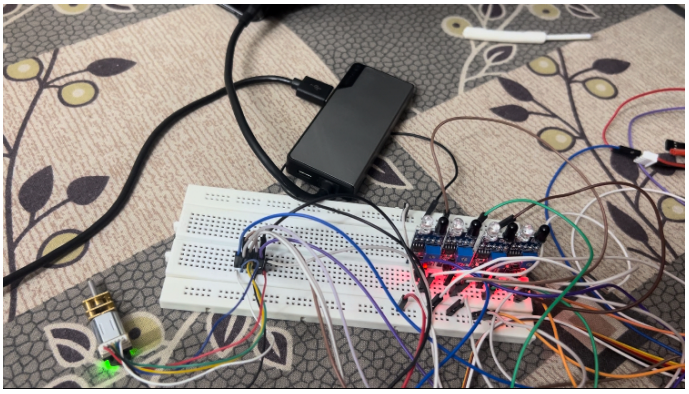


Figure 5: 1st prototype

Overall, the testing and prototyping phase played a pivotal role in refining our design, identifying potential pitfalls, and ultimately laying the groundwork for a successful Micromouse solution.

4 PCB Design

4.1 PCB Schematic

Following the successful testing of our hardware components in the initial prototype, the next step in our development process involved transitioning from a breadboard setup to a more permanent and optimized solution. This entailed making connections and designing a Printed Circuit Board (PCB) schematic using DipTrace, a comprehensive PCB design software.

Careful consideration was given to every aspect of the circuit layout, ensuring efficient routing of signals and power distribution while minimizing interference and signal degradation. Each component, including the ESP32 microcontroller, the Infrared Sensors (IRS), the DC motors with built-in encoders, the L298N motor driver module, and any additional support components, was meticulously integrated into the schematic diagram.

The process involved detailed planning and strategic placement of components to optimize space utilization and facilitate ease of assembly and maintenance. Clear labeling and annotation were incorporated into the schematic to provide clarity and guidance during the subsequent PCB layout and fabrication stages.

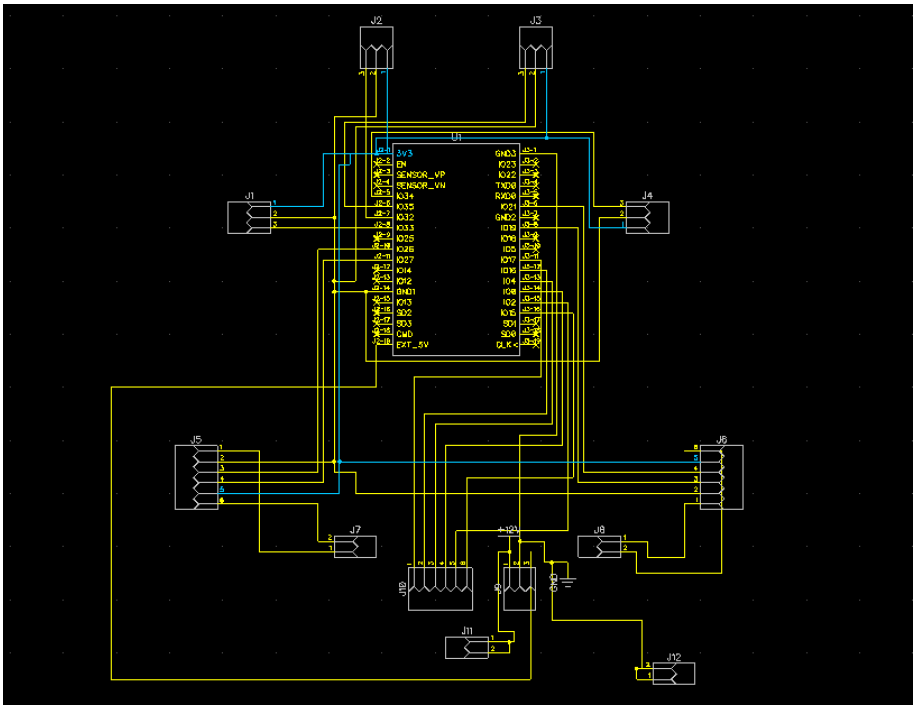


Figure 6: PCB schematic

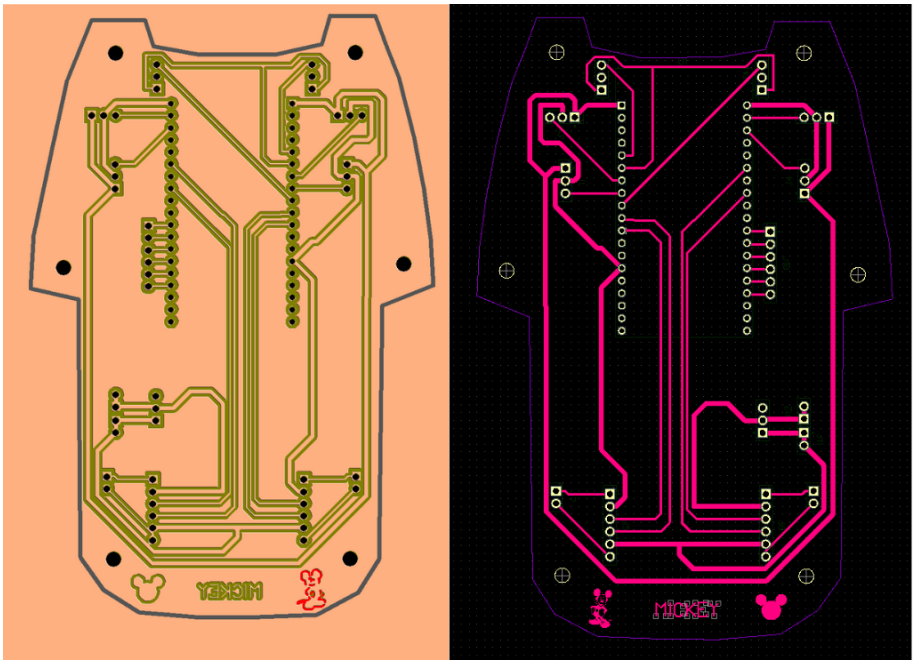


Figure 7: PCB Layout

4.2 PCB Layout

4.3 Printing PCB and Assembly

Once the PCB schematic was finalized using DipTrace, the next crucial step in our hardware development journey was to fabricate the actual Printed Circuit Board (PCB). Utilizing CopperCAM software, we meticulously translated the PCB layout from DipTrace into a format suitable for fabrication.

The layout process involved precise placement of traces, vias, pads, and other elements to ensure optimal signal integrity, power distribution, and component connectivity. Careful attention was paid to routing considerations, such as minimizing signal interference, optimizing trace lengths, and adhering to design rules and constraints.

With the layout finalized, we exported the design files to CopperCAM for processing. CopperCAM enabled us to generate the necessary Gerber files and toolpaths for PCB fabrication. We meticulously reviewed the generated files to ensure accuracy and completeness before proceeding to the fabrication stage.

Using high-quality materials and advanced fabrication techniques, we printed the PCB with precision and attention to detail. The fabrication process involved etching copper layers, drilling holes for component mounting, and applying solder mask and silkscreen for insulation and component labeling, respectively.

Once the PCB was fabricated, we meticulously assembled all hardware components onto the board. Each component, including the ESP32 microcontroller, Infrared Sen-

sors (IRS), DC motors with built-in encoders, and the L298N motor driver module, was carefully soldered onto the PCB following industry-standard assembly practices.

Throughout the assembly process, we ensured proper alignment, orientation, and soldering technique to guarantee the reliability and functionality of the final assembly. Rigorous quality assurance checks were conducted to verify component placement, solder joints, and overall integrity of the assembled PCB.

By meticulously translating our hardware design from concept to reality through the PCB fabrication and assembly process, we brought our Micromouse solution one step closer to completion. This meticulous attention to detail and commitment to excellence underscored our dedication to delivering a high-quality and reliable product.

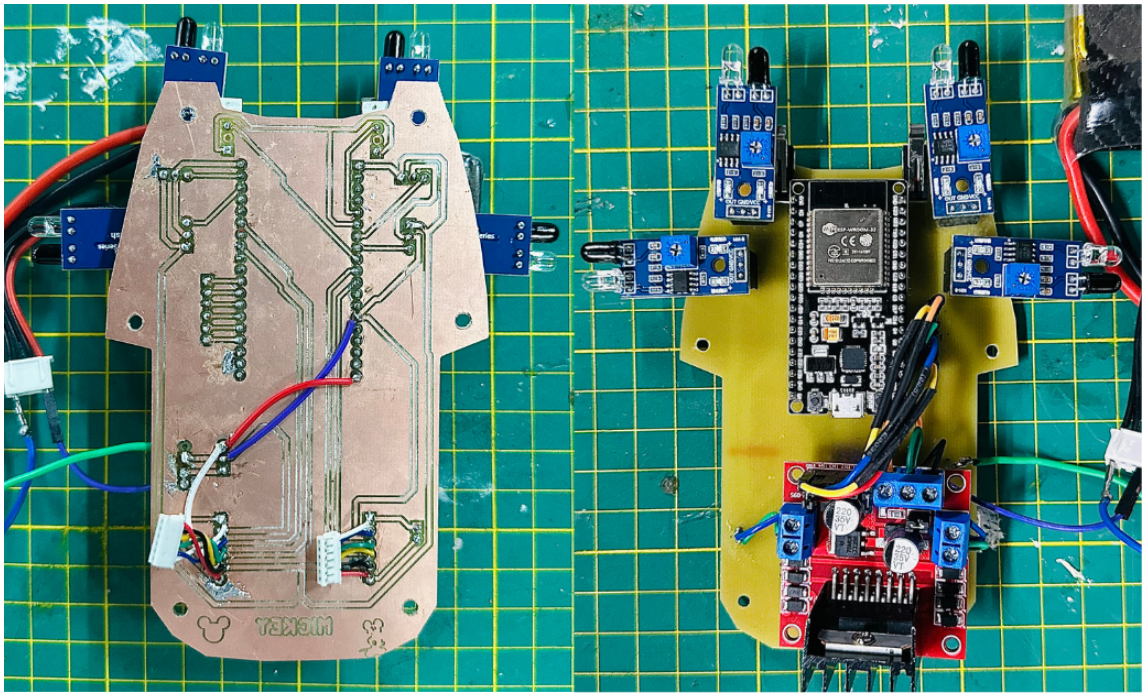


Figure 8: Assembled PCB

5 Calculations:

Calculation of RPM Ratio of Both Motors while Turning: While turning, the motor's RPM Ratio will be equal to the tires' RPM Ratio on the micro-mouse's opposite side. So, let's calculate the RPM Ratio of the tyres while turning.

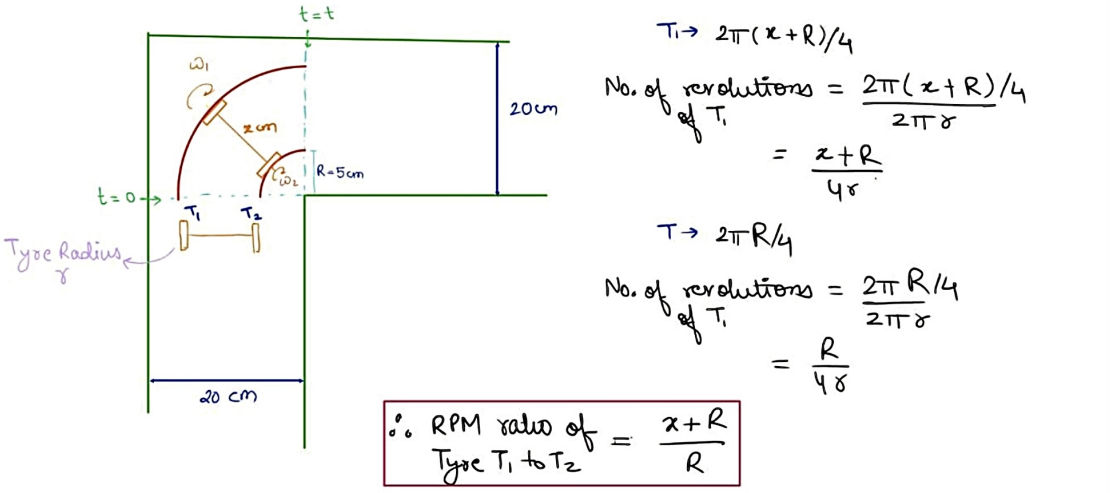


Figure 9: Assembled PCB

Maximal motor turning speed: Reading from the motor data sheet, we know our motors, with a 1:30 gear ratio, have a maximal velocity of $n_max = 1000$ rpm. Our wheel is the 34.4 mm diameter.

$$V_{max} = n_{max}.\pi.d = 1000.\pi.(34.4mm) = 1800mm/s$$

Power Dissipation

Table 1: Power Dissipation across components.

Components	Power dissipatio
ESP32	78.32 mW
IR sensors	25 mW
L298N Motor Driver	20 W

6 Software Overview

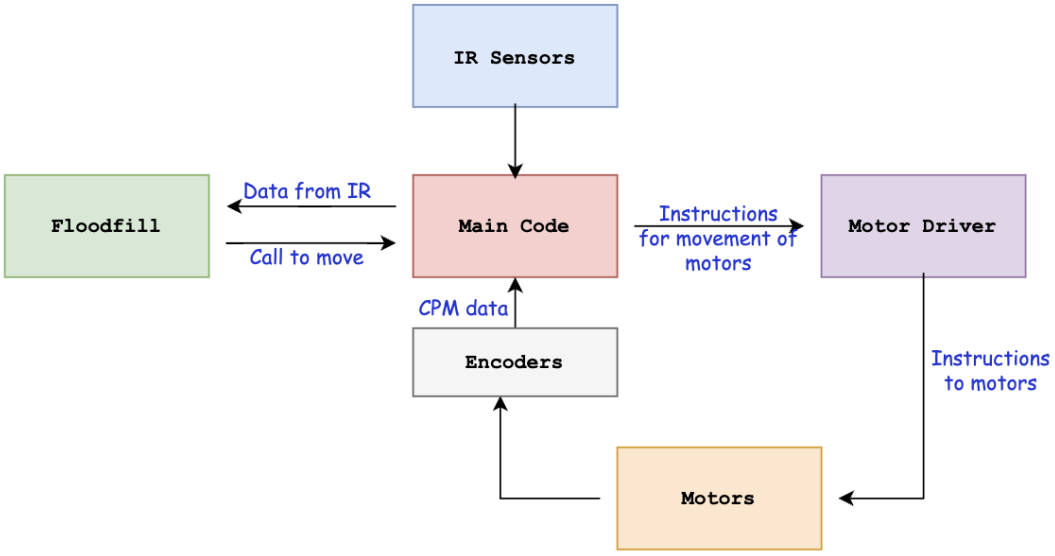


Figure 10: Block Diagram

6.1 Code Implementation

For the purpose of solving the maze, the most common and frequent choice the flood fill algorithm. This project also focuses on the floodfill algorithm to find the next probable square the mouse should go to from the current square in the maze.

6.2 Initialization

For implementation of the floodfill algorithm, have to take three arrays initialization, two 1D arrays for storing the horizontal and vertical walls, and one 2D array to store the flood values of each square of the maze.

6.3 FloodFill Algorithm

1. Define start and destination coordinates
2. Initialize all the arrays with appropriate values
3. Repeat until you reach destination:
   (a) Update flood values and walls
   (b) Find the direction in which the flood values reduces
   (c) Move in that direction
   (d) Update your current position and orientation based on the movement
4. Apply same to return to start from destination coordinate.

6.4 Code Snippets

6.4.1 To move in the direction found using floodfill

```
updateWalls(current, orient);
updateFloodVals(current, destination_cells, num_destinations);
while(flood_vals[current.x][current.y] != 0){
    showText();
    API_setColor(current.x, current.y, color);
    Cell* neighbours= getNeighbours(current);
    bool all_greater = true;
    int min_flood_val = flood_vals[current.x][current.y];
    visitedMaze[current.x][current.y] = 1;
    int dir_to_move = -1;
    int* reachable = (int*)malloc(4*sizeof(int));
    reachable[0] = (current.y<MAZE_MAX_SIZE-1)?!horizontal_walls[
                                    current.x][current.y]:0;
    reachable[1] = (current.x<MAZE_MAX_SIZE-1)?!vertical_walls[
                                    current.x][current.y]:0;
    reachable[2] = (current.y>0)?!horizontal_walls[current.x][
                                    current.y - 1]:0;
    reachable[3] = (current.x>0)?!vertical_walls[current.x - 1][
                                    current.y]:0;
    for(int i = 0; i < 4; i++){
        if(reachable[i]){
            if(flood_vals[neighbours[i].x][neighbours[i].y] <
                                            flood_vals[current.x]
                                            [current.y]){
                all_greater = false;
                if(flood_vals[neighbours[i].x][neighbours[i].y] <
                                                min_flood_val){
                    min_flood_val = flood_vals[neighbours[i].x][
                                                    neighbours[i]
                                                    .y];
                    dir_to_move = i;
                }
            }
        }
    }
    switch(orient){
        case NORTH:
            switch(dir_to_move)
            {
                case NORTH: API_moveForward(); current.y++; break;
                case EAST: API_turnRight(); orient = EAST;
                                                API_moveForward()
                                                ; current.x++;
                                                break;
                case WEST: API_turnLeft(); orient = WEST;
                                                API_moveForward()
                                                ; current.x--;
                                                break;
                case SOUTH: API_turnRight(); API_turnRight();
                                                orient = SOUTH;
                                                API_moveForward()
```

```
                                                ; current.y--;
                                                break;
            default: break;
        }break;
    case EAST:
        switch(dir_to_move)
        {
            case EAST: API_moveForward(); current.x++; break;
            case SOUTH: API_turnRight(); orient = SOUTH;
                                                API_moveForward()
                                                ; current.y--;
                                                break;
            case NORTH: API_turnLeft(); orient = NORTH;
                                                API_moveForward()
                                                ; current.y++;
                                                break;
            case WEST: API_turnRight(); API_turnRight(); orient
                                                 = WEST;
                                                API_moveForward()
                                                ; current.x--;
                                                break;
            default: break;
        }break;
    case SOUTH:
        switch(dir_to_move)
        {
            case SOUTH: API_moveForward(); current.y--; break;
            case WEST: API_turnRight(); orient = WEST;
                                                API_moveForward()
                                                ; current.x--;
                                                break;
            case EAST: API_turnLeft(); orient = EAST;
                                                API_moveForward()
                                                ; current.x++;
                                                break;
            case NORTH: API_turnRight(); API_turnRight();
                                                orient = NORTH;
                                                API_moveForward()
                                                ; current.y++;
                                                break;
            default: break;
        }break;
    case WEST:
        switch(dir_to_move)
        {
            case WEST: API_moveForward(); current.x--; break;
            case NORTH: API_turnRight(); orient = NORTH;
                                                API_moveForward()
                                                ; current.y++;
                                                break;
            case SOUTH: API_turnLeft(); orient = SOUTH;
                                                API_moveForward()
                                                ; current.y--;
                                                break;
```

```
                case EAST: API_turnRight(); API_turnRight(); orient
                                                    = EAST;
                                                API_moveForward()
                                                ; current.x++;
                                                break;
                default: break;
            }break;
        default: break;
    }
    updateWalls(current, orient);
    updateFloodVals(current,destination_cells, num_destinations);
}
```

6.4.2 To update flood values using flood fill

```
void updateFloodVals(Cell current, Cell* destination_cells, int
                                num_destinations) {
    if(current.x == destination_cells[0].x && current.y ==
                                    destination_cells[0].y &&
                                    flood_vals[current.x][current
                                    .y] == flood_vals[
                                    destination_cells[0].x][
                                    destination_cells[0].y]){
        flood_vals[current.x][current.y] = 0;
        return;
    }
    for (int i = 0; i < MAZE_MAX_SIZE; i++) {
        for (int j = 0; j < MAZE_MAX_SIZE; j++) {
            flood_vals[i][j] = INF;
        }
    }

    for (int d = 0; d < num_destinations; d++) {
        Cell dest_cell = destination_cells[d];
        flood_vals[dest_cell.x][dest_cell.y] = 0;
    }

    bool changed = true;
    while (changed) {
    changed = false;
    for (int i = 0; i < MAZE_MAX_SIZE; i++) {
        for (int j = 0; j < MAZE_MAX_SIZE; j++) {
            if (flood_vals[i][j] != INF) {
                Cell current = {i, j};
                Cell* neighbours = getNeighbours(current);
                for (int orient = 0; orient < 4; orient++) {
                Cell neighbour = neighbours[orient];
                if (neighbour.x >= 0 && neighbour.x < MAZE_MAX_SIZE
                                                    &&
                neighbour.y >= 0 && neighbour.y < MAZE_MAX_SIZE &&
                !isWall(current, orient)) {
                int new_dist = flood_vals[current.x][current.y] + 1
                                                    ;
```

```
                if (new_dist < flood_vals[neighbour.x][neighbour.y]
                                                    ) {
                    flood_vals[neighbour.x][neighbour.y] = new_dist
                                                    ;
                    path_info[neighbour.x][neighbour.y].prev =
                                                    current; //
                                                    Update
                                                    previous cell
                    path_info[neighbour.x][neighbour.y].dist =
                                                    new_dist;
                    changed = true;
                }
                }
                }
                free(neighbours); // Free the allocated memory for
                                                    neighbours
            }
        }
    }
}
char text[20];
sprintf(text,"previous neighbour: %d %d", path_info[current.x][
                                    current.y].prev.x, path_info[
                                    current.x][current.y].prev.y);

log_it(text);
}
```

6.4.3 Function to update walls based on the current cell and the position of walls

```
void updateWalls(Cell current, int orient){
switch(orient){
    case NORTH:
        if(API_wallFront()){
            horizontal_walls[current.x][current.y] = 1;
        }if(API_wallRight()){
            vertical_walls[current.x][current.y] = 1;
        }if(API_wallLeft()){
            if(current.x>0)
                vertical_walls[current.x - 1][current.y] = 1;
        }break;
    case EAST:
        if(API_wallFront()){
            vertical_walls[current.x][current.y] = 1;
        }if(API_wallRight()){
            if(current.y>0)
                horizontal_walls[current.x][current.y-1] = 1;
        }if(API_wallLeft()){
            horizontal_walls[current.x][current.y] = 1;
        }break;
    case SOUTH:
        if(API_wallFront()){
            if(current.y>0)
```

```
                horizontal_walls[current.x][current.y-1] = 1;
            }if(API_wallRight()){
                if(current.x>0)
                    vertical_walls[current.x - 1][current.y] = 1;
            }if(API_wallLeft()){
                vertical_walls[current.x][current.y] = 1;
            }break;
        case WEST:
            if(API_wallFront()){
                if(current.x>0)
                    vertical_walls[current.x - 1][current.y] = 1;
            }if(API_wallRight()){
                horizontal_walls[current.x][current.y] = 1;
            }if(API_wallLeft()){
                if(current.y>0)
                    horizontal_walls[current.x][current.y-1] = 1;
            }break;
        default: break;
    }
    return;
    }
```

6.5 Simulation

Given snapshot of simulation shows how the bot is travelling the maze and turning on detecting walls and reaches destination. All the visited cells are represented as Green coloured, destination cells are of blue coloured and rest of the cells are of black colour which are not yet discovered. The walls corresponding to the visited cells are also updated in the arrays for horizontal and vertical walls.

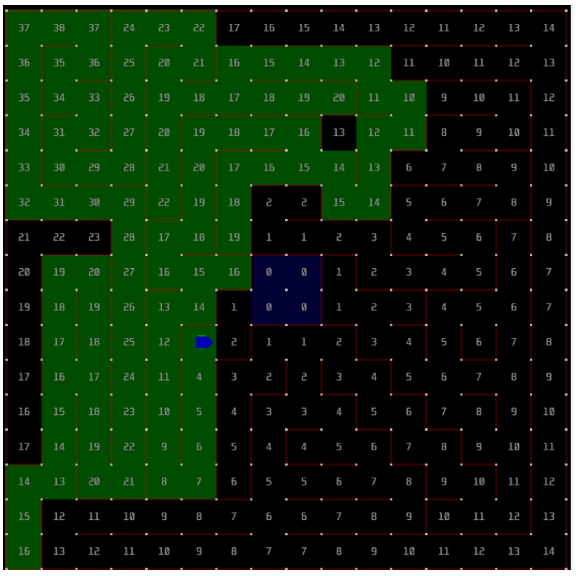


Figure 11: Bot in Virtual environment

7 Final Prototype

Following the assembly, we uploaded the necessary code onto the microcontroller to facilitate dry runs and preliminary testing. This code was designed to control the movement of the Micromouse, interpret sensor data, and navigate through a simulated maze environment.

During the dry runs, we closely monitored the performance of the Micromouse, evaluating its ability to maneuver through the maze, detect obstacles, and make navigational decisions. Any discrepancies or issues observed during testing were meticulously documented, and adjustments were made as necessary to optimize performance.

Through iterative testing and refinement, we fine-tuned the code and hardware configuration to enhance the Micromouse's performance and reliability.

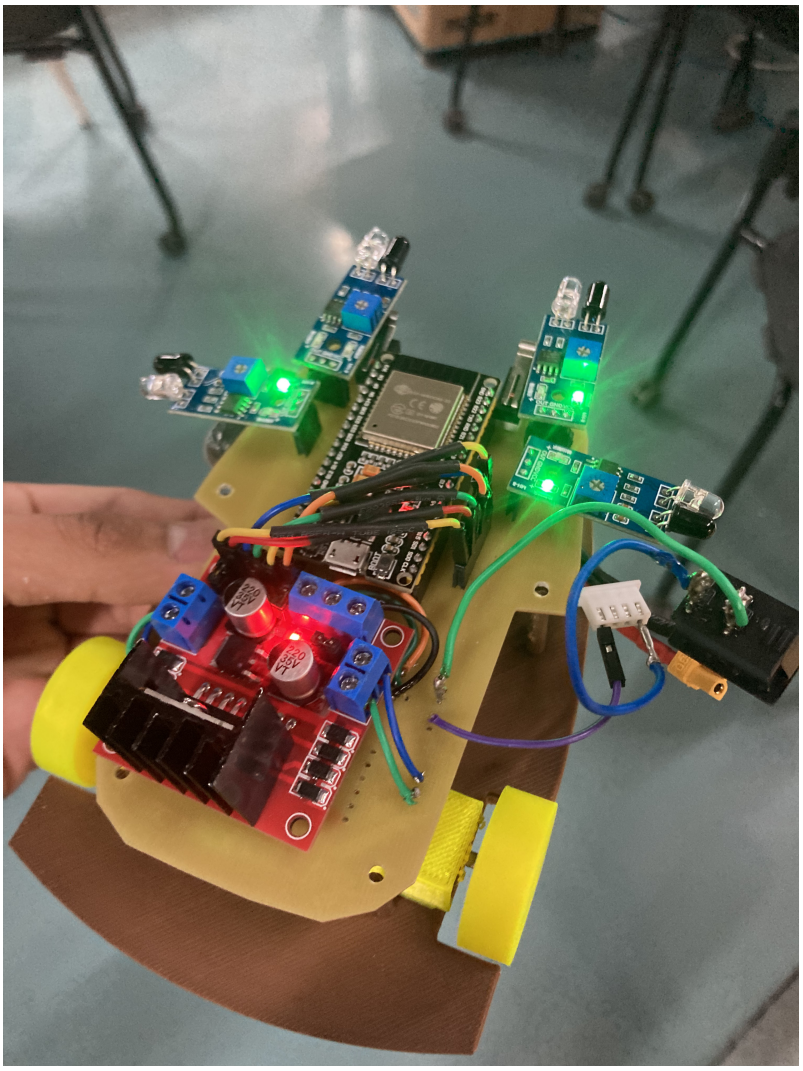


Figure 12: Final Prototype

8 Drawbacks

- Due to time constraints, we were not able to implement the floodfill algorithm on the ESP32 micro processor
- Not able to perform enough number of test and trials of the prototype for callibration.

9 Acknowledgement

We want to express our appreciation to Prof. Anirudh Mali and TA Mr. Pupul Dalbehera for their valuable guidance and assistance during the weekends and late nights. Their expertise and support were instrumental in the successful completion of the project. Thank you for sharing your knowledge and contributing to our learning experience. We also thank IIT Gandhinagar and Maker bhawan for providing us with this course.

References

[1] `http://ieeexplore.ieee.org/abstract/document/5971240/.`

[2] `https://www.instructables.com/Micro-Mouse-for-Beginnersth/`

[3] `https://github.com/Isuru-Dissanayake/piccola`