

Table of Contents

Contents

1	Progress Report	2
1.1	1 st Iteration	2
1.2	2 nd Iteration	2
1.3	3 rd Iteration	2
1.4	4 th Iteration	2
2	Mechanical Design of MazeBot	3
2.1	Top Level Mechanical Structure and Specifications	3
2.2	First Iteration	4
2.3	Second Iteration	6
3	Software Design of MazeBot	8
3.1	Variables Used to Define the Position of the Robot in the Maze and the Size of the Maze	9
3.2	Constants and Variables Used for Representation of Directions	10
3.3	Constants Used for Display	10
3.4	Constants Used for Moving Mechanism	11
3.5	Constants Used for Representation of Wall	12
3.6	Constants Used for Beeping Mechanism	12
3.7	Displaying Function	13
3.8	Moving Forward Function	14
3.9	Turning Functions	15
3.10	Wall Detecting Function	16
3.11	Function for Storing Data of the Walls	17
3.12	Functions for setting up the direction that need to be used	19
3.13	Functions for Finding Existence of Wall from the data	20
3.14	Functions for Readjusting in Certain Directions	21
3.15	Function for Movement All Together	24
3.16	Functions for Returning Algorithm	25
3.17	Main Function	28
4	Appendix	30
4.1	Full Source Code	30

1 Progress Report

1.1 1st Iteration

Should be achieved by: July 12, 2016

- Robot moves forward exactly one cell length and turns 90 ° accurately
This is the most important feature of our robot. By ensuring accurate movement, we limit the need to readjust in each cell.
- Robot follows right wall
This is the algorithm needed to find the unique solution to the maze. Once movement is implemented, we can just build on top of it to collect more data as we move through the maze.

1.2 2nd Iteration

Should be achieved by: July 12, 2016

- Robot tracks its orientation and location in maze
This is dependency for most of the other requirements. By tracking movement in the cell, we are able to know when we have reached our goal, how we reached it (in order to implement coming back shortest path) and where to store cell information in our 2-D array.
- Robot beeps when reached target
- Robot stores wall information, visited/unvisited status and orientation at entry in 2-D array.
This is required for more advanced algorithm features we wish to implement such as having the robot not check the same wall twice. Furthermore, this was required in order to display current location and wall information graphically. We chose to implement it now because it was a major criteria requirement.

1.3 3rd Iteration

Should be achieved by: July 19, 2016

- Robot returns with shortest path
Since we know the route we took to get to our final location, we can now implement the cancelling algorithm (described in detail in **Section 3.16**).
- Robot displays current location graphically on screen
We implemented this now so that we could more easily test our algorithm. The last project feature we wanted to implement was to not check the same wall twice. We thought that this would be difficult to implement and as such improved our debugging capabilities before continuing.

1.4 4th Iteration

Should be achieved by: July 26, 2016

- Improve algorithm such that robot doesn't check same wall twice
The premise behind this optimization is that imagine that we are in a cell with a wall to the South. If we've checked that there is a wall toward the South, we know that in the cell below our current cell, there is a wall to the North. The most general case will be programmed and described in **Section 3**. This has two major advantages: we do not have to waste time turning and we do not incur extra error because of unneeded turning.

2 Mechanical Design of MazeBot

2.1 Top Level Mechanical Structure and Specifications

Our robot needed very accurate movement in order to be successful in the maze. This criteria depends heavily on whether or not the motor encoders report accurate values to the algorithm. In order to achieve this, we had to ensure:

- The wheels do not slip
- The robot does not hit walls
- The drive system is sturdy
- The gears are securely held in place and make proper contact.

However, there is a limit to how much we can do to minimize mechanical error in movement. As a result, we will have to readjust after a certain number of cells. This is done by driving into the wall, turning 90° and driving into the wall again. However, having to readjust too often is problematic as this adds time to our average time in each cell which is a major criteria point. As a result, we hope to minimize the amount of times we needed to readjust by maximizing the accuracy of the mechanical system.

2.2 First Iteration

- **Goals:**

- Able to go 3 cells without needing to readjust
- Able to turn 90° accurately.

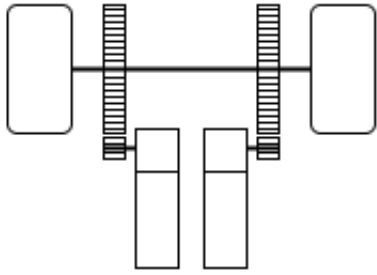


Figure 1: First Iteration Drive System

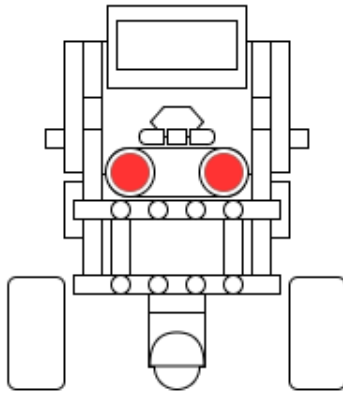


Figure 2: First Iteration Front View

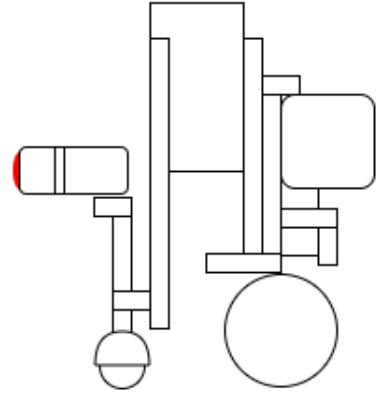


Figure 3: First Iteration Side View

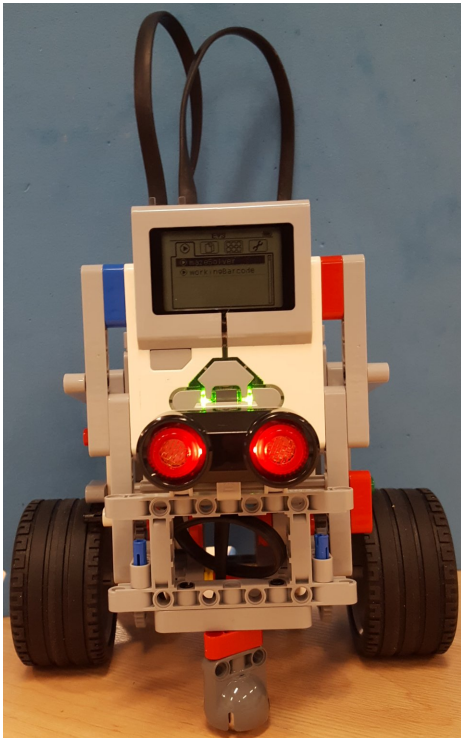


Figure 4: First Iteration Front View

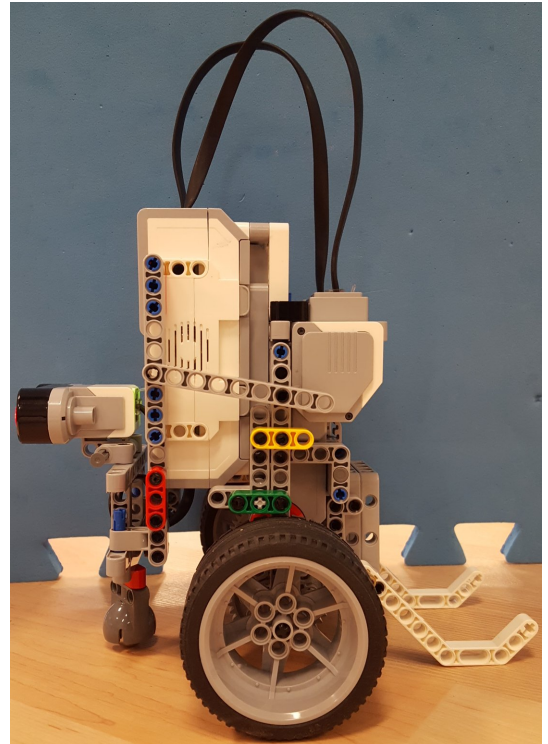


Figure 5: First Iteration Side View

- **Observations & Measurements**

- **Goal:** Able to go 3 cells without needing to readjust
- **Failed:** Needed to readjust every second cell
- **Goal:** Able to turn 90° accurately.

Test - An error of even $\pm 1^\circ$ will cause accumulate to a significant error when traversing cells. However, error in turning is hard to notice. Therefore, we chose to have robot turn 90° 8 times in place in order to propagate any error significantly.

- **Failed:** Robot had error of $\pm 20^\circ$

- **Reasons for Test Failures**

- **Structural Integrity of the Drive System**

We are unable to find space to properly secure the left and right drive wheels. When testing, we found one wheel to slipped forward and the other to slipped back when turning which defeats the accuracy of the encoder. Because of this, we are unable to meet our goal of accurate movement.

- **Robot is too large.**

Since the brick is upright, it is top heavy. We needed two rods in the back and one metal ball in the front in order to balance the robot. The additions of the two rods and one metal ball negates the spacial advantage of having the robot's brick be upright. Even though the dimensions of the robot are within the size of one square, it leaves very little room for error. As such, the robot begins to run into walls after the 2nd turn.

- **Wheels are too big**

In order for the motor encoder to be accurate, the wheels must not slip. In order to maximize friction, we decided to use the largest wheels in the set. However. the extra friction with the ground from the larger wheels is not worth the extra size added to the robot. When we replaced the large wheels with smaller wheels, we noticed little to no change in accuracy of movement.

- **Conclusion**

In conclusion, we have decided to no longer have our robot upright. This will allow us to have enough room to properly secure the drive system. Which will allow us to ensure that the gears make proper contact and do slide forward or backward. This ensures maximum encoder accuracy. By having the robot level with the table, we will be able to take out the additional support that we needed before to hold the bot upright. This will allow more room for error when turning and going into new cells.

2.3 Second Iteration

- **Goals:**

- Able to go 3 cells without needing to readjust
- Able to turn 90° accurately.

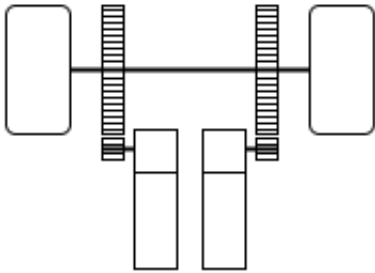


Figure 6: Second Iteration Drive System

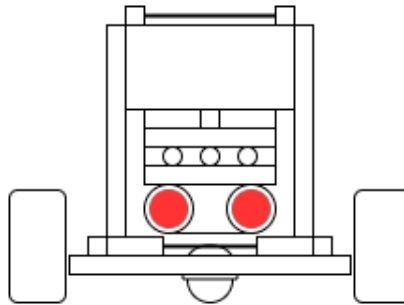


Figure 7: Second Iteration Front View

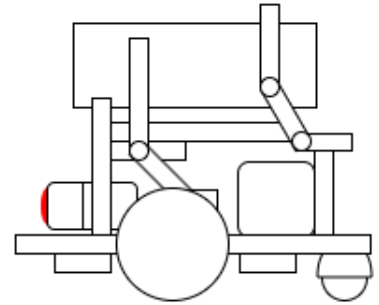


Figure 8: Second Iteration Side View

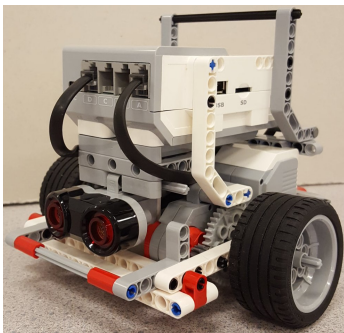


Figure 9: Second Iteration Drive System

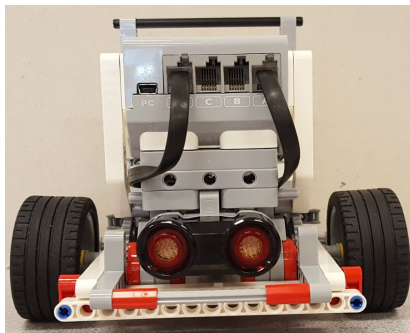


Figure 10: Second Iteration Front View

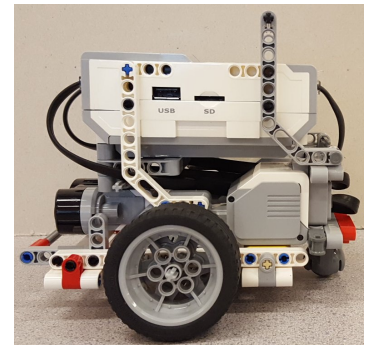


Figure 11: Second Iteration Side View

- **Observations & Measurements**

- **Goal:** Able to go 3 cells without needing to readjust

- **Passed:** Needed to readjust every fifth cell

- **Goal:** Able to turn 90° accurately.

- Test - An error of even $\pm 1^\circ$ will cause accumulate to a significant error when traversing cells. However, error in turning is hard to notice. Therefore, we chose to have robot turn 90° 8 times in place in order to propagate any error significantly.

- **Passed:** Robot had an unnoticeable error even after eight turns

- **Reasons for Test Successes**

Our hypotheses were correct. By securing the gears, we were able to make the motor encoders much more accurate and as a result, have the robot move in much more controlled way. Furthermore, having the robot much more compact allowed the system to have a larger tolerance for error.

- **Conclusion**

After much contemplation, we have decided that this is the best design. The need to readjust cannot be avoided because of the uncertainty in turning caused by the legos flexing and backlash in the gears. In order to further reduce the error, we have decided to make the algorithm for the robot as efficient as possible. An example of this is to avoid turning to check for walls as much as possible because turning is our least accurate movement.

3 Software Design of MazeBot

The main goal with the software of the mazebot was to create program that solved the problem simply and was easy to build upon. Furthermore, we wanted our software to have very few constants that we would need to tested for. For example, in order to move forward one cell, we would need to give the following function the degrees to move each of our drive motors:

```
setMotorSyncEncoder(leftDrive, rightDrive, 0, Degrees, BACKWARD);
```

The degrees needed to move one cell forward could be achieved by constantly testing different values of degrees to achieve the movement to the new cell. However, we chose to calculate the exact degrees that the robot's drive motors would need to move in order to move exactly one cell forward. This approach in contrast to the former has two advantages:

1. It allows us to isolate any problems with moving accurately to a mechanical problem.
2. We would not have an accumulation of error because of us testing incorrectly.

Therefore, we chose to mathematically calculate the degrees that we needed to move the motors rather than testing.

A sketch of the derivation of how many degrees to move forward is shown below:
Therefore:

```
degrees = (SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL) * DRIVE_GEAR_RATIO * ONE_ROTATION
```


3.1 Variables Used to Define the Position of the Robot in the Maze and the Size of the Maze

- Two constant that represent the initial position of the robot in the maze were declared. These will be entered when we begin our demo.

```
int const START_ROW = ;  
int const START_COL = ;
```

- Two constants that represent the target position in the maze were declared. These will be entered when we begin our demo.

```
int const END_ROW = ;  
int const END_COL = ;
```

- Two global variables that represent the current position of the robot in the maze were declared. These are initialized as the starting position.

```
int currentRow = START_ROW;  
int currentCol = START_COL;
```

- An array that represents the orientation that the bot has as it enters each cell was defined. The size of the array is four times larger than the product of the maze width and maze height because the maximum amount of times that the robot can go into each cell is four times (worst case scenario).

```
int entered[MAZE_WIDTH*MAZE_HEIGHT*4];  
int lastEnteredIdx = 0;
```

- A constant that represents the dimension of a single cell was defined

```
float const SIZE_OF_ONE_CELL = 22.5425; // in cm
```

- Four constants that represent the size of the maze were declared

```
int const MAZE_WIDTH = 4;  
int const MAZE_HEIGHT = 6;  
int const LAST_MAZE_HEIGHT_INDEX = MAZE_HEIGHT - 1;  
int const LAST_MAZE_WIDTH_INDEX = MAZE_WIDTH - 1;
```

3.2 Constants and Variables Used for Representation of Directions

- The four constants that represent each of the directions were declared:

```
#define NORTH 0
#define EAST 1
#define SOUTH 2
#define WEST 3
```

- A structure named cell was declared and it has five parameters. This track where the walls are, what direction we entered from and whether we have visited the cell.

```
typedef struct{
    int NWall;
    int SWall;
    int EWall;
    int WWall;
    char Visited;
    int entryDir;
}cell;
```

- A 2-D array called "Maze" with the data type cell was declared. This data type is described above.

```
cell Maze[MAZE_HEIGHT][MAZE_WIDTH];
```

3.3 Constants Used for Display

- Two constants that represent the size of the screen width and height were defined

```
#define SCREEN_HEIGHT 127
#define SCREEN_WIDTH 177
```

- Two constants that represent the each cell's size on the screen were defined

```
#define CELL_HEIGHT (SCREEN_HEIGHT / MAZE_HEIGHT)
#define CELL_WIDTH (SCREEN_WIDTH / MAZE_WIDTH)
```

- Two constants are defined which represent the robot's position in each cell in the screen

```
#define CELL_HEIGHT_MIDDLE (CELL_HEIGHT / 2)
#define CELL_WIDTH_MIDDLE (CELL_WIDTH / 2)
```

3.4 Constants Used for Moving Mechanism

- When we calculated the degrees to move the encoder, we had two contributing errors that caused the motors to move less than they needed to. First of all, we were using integer division to find the degrees to move the motors. Therefore, the remainder is truncated and this causes the robot to move less than one cell or less than 90° . Similarly, the PID control caused the robot to move less than the desired target. Therefore, three constants were declared which are added to the encoder input values and only needed to be tested once in order to supplement the errors.

```
float const UNCERTAINTY_STRAIGHT = 23;
float const UNCERTAINTY_ROT = 28;
float const UNCERTAINTY_READJUST = 35;
```

- Back and forward speed of the motors were defined with constants for simplification of the code.

```
int const FORWARD = -100;
int const BACKWARD = -FORWARD;
```

- Encoder input constants were declared

```
float const ONE_ROTATION = 360 + UNCERTAINTY_STRAIGHT;
float const QUARTER_ROTATION = 180 + UNCERTAINTY_ROT;
float const DRIVE_GEAR_RATIO = 5;
float const DIAMETER_OF_WHEEL = 5.5; // in cm
float const CIRCUMFERENCE_OF_WHEEL = PI * DIAMETER_OF_WHEEL;
```

- The amount of time that the bot will drive into the wall in order to readjust was defined. Timing algorithm was used because the flat surface at the front of the robot adjusts the bot as it drives into the wall.

```
int const MILLISECS_TO_DRIVE_INTO_WALL = 1100;
```

- A constant that represents how often the robot has to readjust its direction was defined. A global variable that increases every time the robot goes into new cells to count for readjust was also defined.

```
int const CELLS_TO_READJUST_AFTER = 3;
int timesForwardWithoutReadjust = 0;
```

3.5 Constants Used for Representation of Wall

- A constant which represent the maximum distance possible between the robot and an object for the robot to consider it a wall.

```
float const DIST_BETWEEN_BOT_AND_WALL = 7.6;
```

- Three constants were defined that represent the robot's knowledge of whether or not there is a wall.

```
#define NOT_PRESENT 0  
#define PRESENT 1  
#define UNKNOWN 2
```

3.6 Constants Used for Beeping Mechanism

- A constant which represent the time and the frequency of the beep when the robot found the target

```
int const MILLI_TO_BEEP_FOR = 200;  
int const FREQUENCY = 300;
```

3.7 Displaying Function

- Function for displaying information about the robot's orientation and location on the screen as well as what it knows about the maze.

```
void drawInfo(int direction);
```

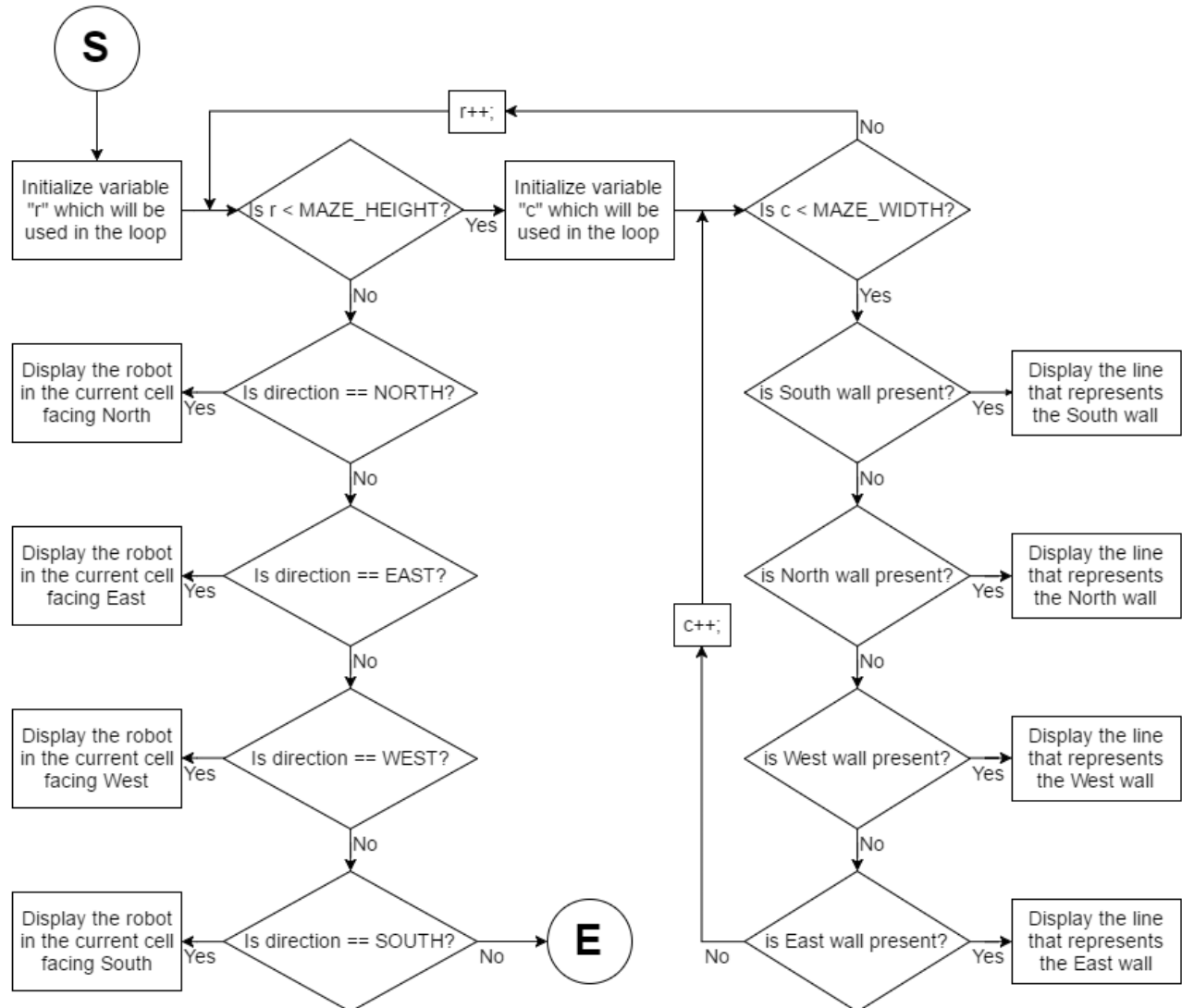


Figure 12: Flow Chart for Displaying Function

- The local variable `direction` is passed into the function but it does not return any variable
- Global variables and constants used are

```
MAZE_WIDTH
MAZE_HEIGHT
CELL_WIDTH
CELL_HEIGHT
CELL_WIDTH_MIDDLE
CELL_HEIGHT_MIDDLE
Maze [] []
```

3.8 Moving Forward Function

- This function moves the the robot forward exactly one cell. It also stores the cell information in the maze array. Finally, it increments how many cells it has moved without readjust.

```
void goFwdCell(int direction);
```

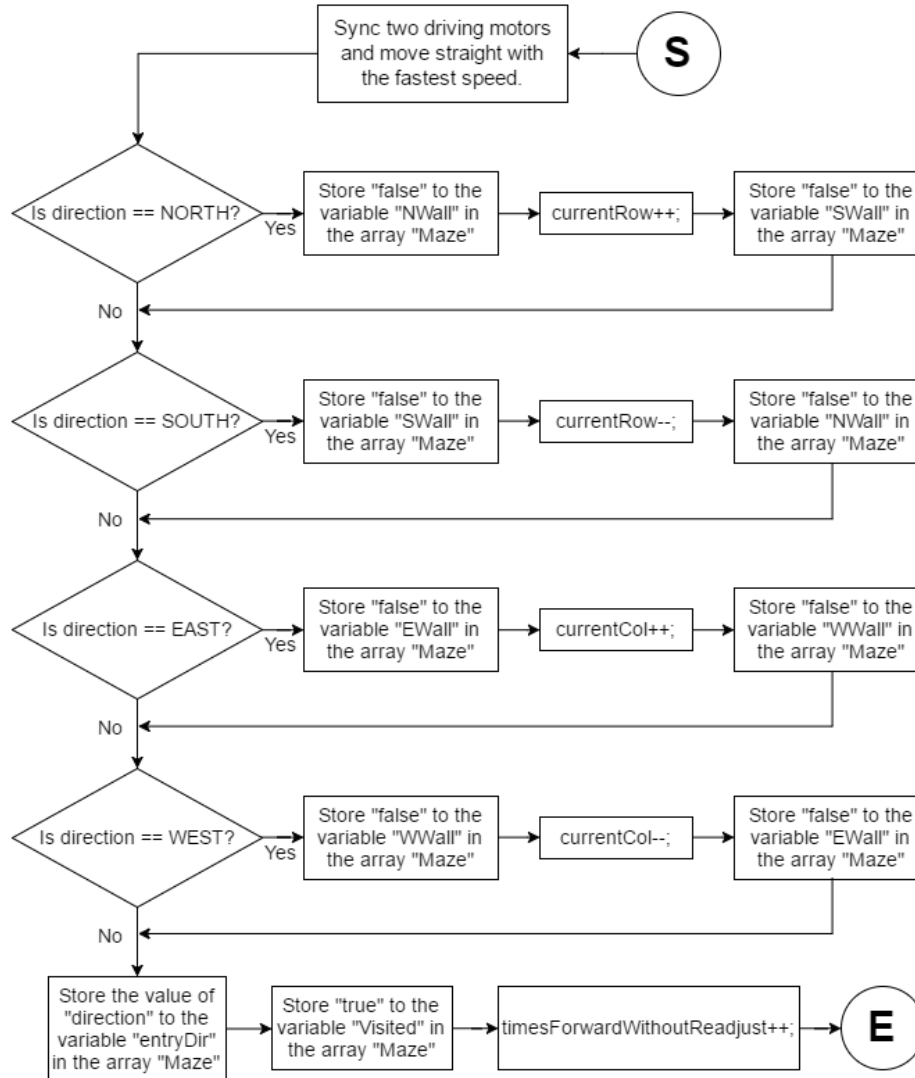


Figure 13: Flow Chart for Moving Forward

- The local variable direction is passed into the function but it does not return any variable
- Global variables and constants used are

```

SIZE_OF_ONE_CELL
CIRCUMFERENCE_OF_WHEEL
DRIVE_GEAR_RATIO
ONE_ROTATION
FORWARD
timesForwardWithoutReadjust
Maze [] []

```

3.9 Turning Functions

- Function for turning right.

```
int Turn90CW(int direction);
```

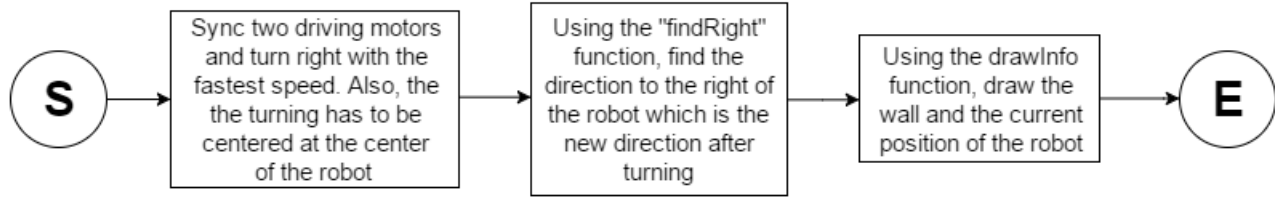


Figure 14: Flow Chart for Turning Right

- The local variable direction is passed into the function. The function returns the new direction.
- Global variables and constants used are

```
QUARTER_ROTATION;  
DRIVE_GEAR_RATIO;  
FORWARD;
```

- This function calls the other functions

```
int findRight(int direction);  
int drawInfo(int direction);
```

- Function for turning left

```
Turn90CW(int direction);
```

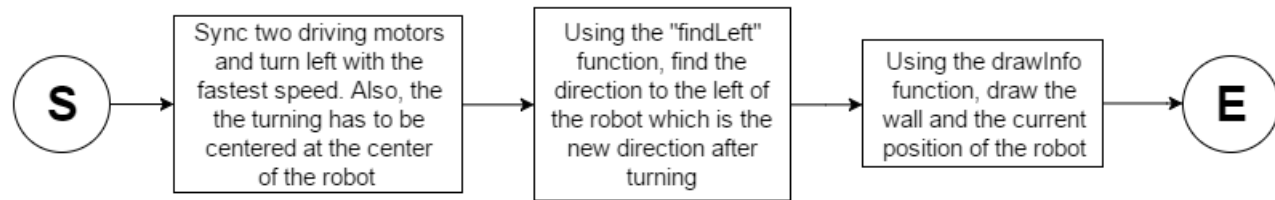


Figure 15: Flow Chart for Turning Left

- The local variable direction is passed into the function. The function returns the new direction.
- Global variables and constants used are

```
QUARTER_ROTATION;  
DRIVE_GEAR_RATIO;  
FORWARD;
```

- This function calls the other functions

```
int findLeft(int direction);  
int drawInfo(int direction);
```

3.10 Wall Detecting Function

- Function that returns whether or not there is a wall in front of the bot.

```
int thereIsWall();
```

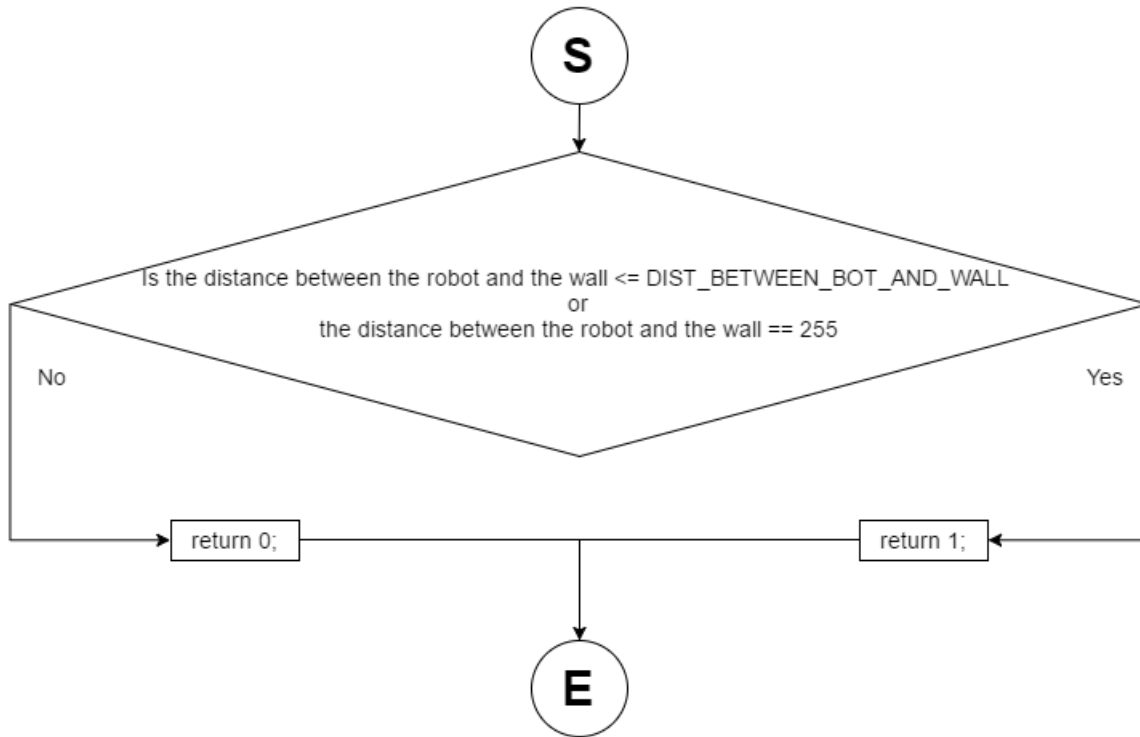


Figure 16: Flow Chart for Wall Detecting Function

- Very simple fuction that returns 1 if the sensor detects the wall
- Global variables and constants used are
DIST_BETWEEN_BOT_AND_WALL

3.11 Function for Storing Data of the Walls

- Function that writes whether there is a wall in the direction the robot is currently facing into the maze array.

```
void writeWall(int direction);
```



Figure 17: Flow Chart for Storing Data Function

- The local variable direction is passed into the function but it does not return any variable.
- Global variables and constants used are:

```

NORTH
SOUTH
EAST
WEST
currentRow
currentCol
PRESENT
LAST_MAZE_HEIGHT_INDEX
LAST_MAZE_WIDTH_INDEX
maze [] []
  
```

- This function calls the other functions

```
int thereIsWall();
```

3.12 Functions for setting up the direction that need to be used

- Function that takes in a direction of the robot and returns the direction towards the back.

```
int findBackDir(int currentDirection);
```

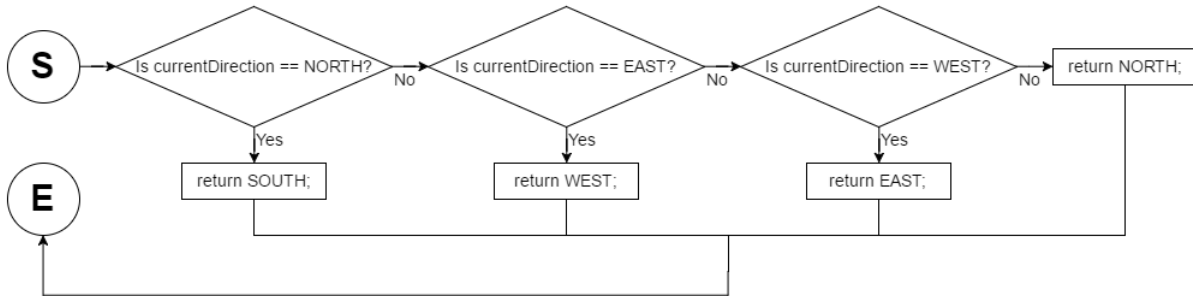


Figure 18: Flow Chart for Finding Back Function

- Function that takes in a direction of the robot and returns the direction to the right of the robot.

```
int findRight(int currentDirection);
```

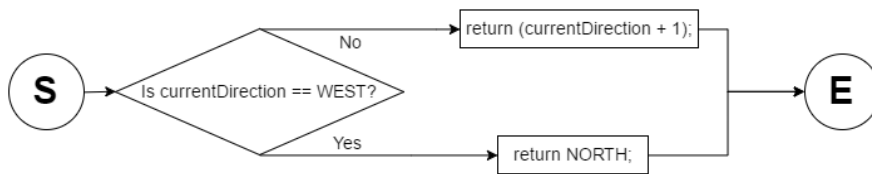


Figure 19: Flow Chart for Finding Right Function

- Function that takes in a direction of the robot and returns the direction to the left of the robot.

```
int findLeft(int currentDirection);
```

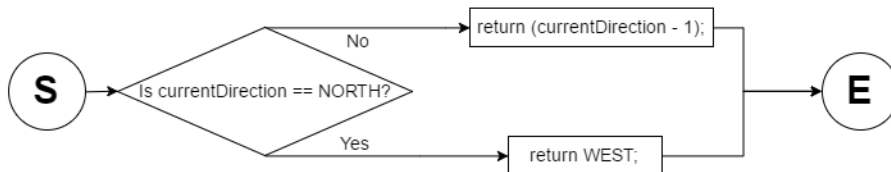


Figure 20: Flow Chart for Finding Left Function

– Global constants used are

NORTH
SOUTH
EAST
WEST

3.13 Functions for Finding Existence of Wall from the data

- Function takes in a direction and returns whether or not there is a wall in that direction from maze array.

```
int isThereWallInDir(int wallDir);
```

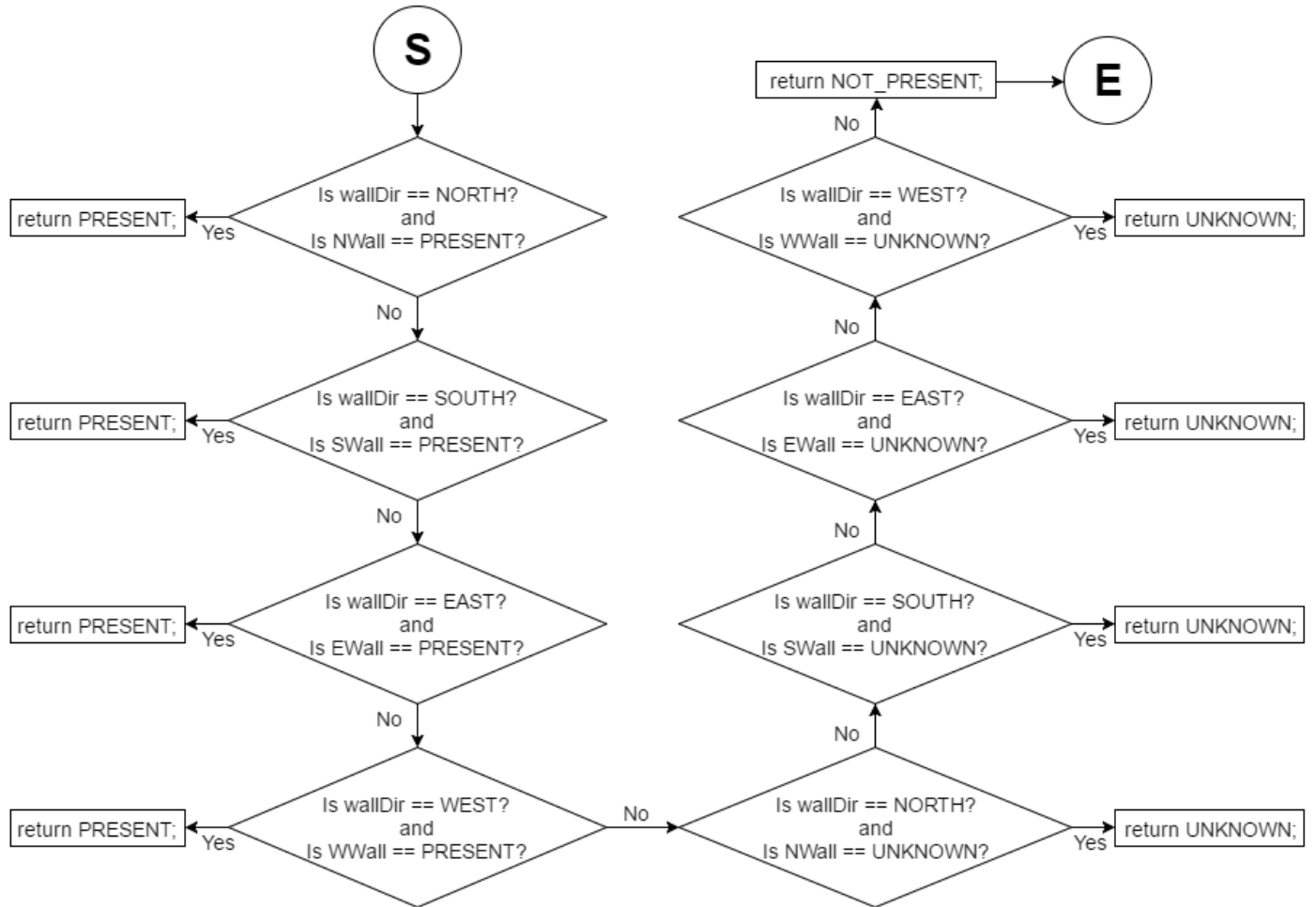


Figure 21: Flow Chart for Finding Existence of Wall from the data

- Global variables and constants used are

```

NORTH
SOUTH
EAST
WEST
PRESENT
UNKNOWN
NOT_PRESENT
maze[] []
  
```

3.14 Functions for Readjusting in Certain Directions

- Function that readjusts robot's position by driving into the wall and coming back to the center of the cell. For this function particularly, we readjust using walls to the front and to the right.

```
void reAdjustCW(int direction);
```

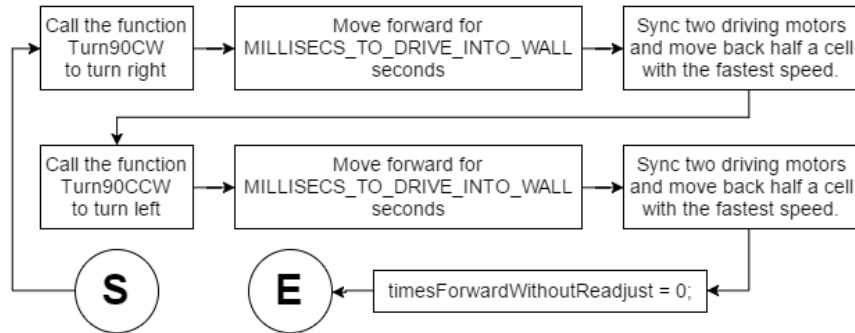


Figure 22: Flow Chart for Readjusting using Front wall and Right wall

- Function that readjusts robot's position by driving into the wall and coming back to the center of the cell. For this function particularly, we readjust using walls to the back and to the left.

```
void reAdjustCCW(int direction);
```

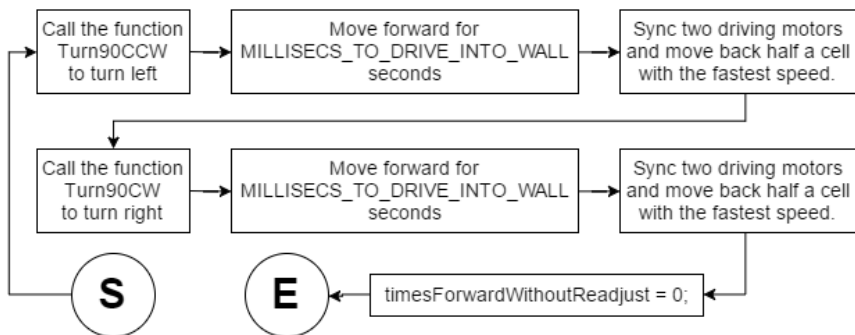


Figure 23: Flow Chart for Readjusting using Back wall and Left wall

- The local variable `direction` is passed into the function but it does not return any variable
- Global variables and constants used are

```

FORWARD
BACKWARD
SIZE_OF_ONE_CELL
CIRCUMFERENCE_OF_WHEEL
DRIVE_GEAR_RATIO
ONE_ROTATION
UNCERTAINTY_READJUST
MILLISECS_TO_DRIVE_INTO_WALL
timesForwardWithoutReadjust
  
```

- This function calls in other functions

```

int Turn90CW(int direction);
int Turn90CCW(int direction);

```

- Function that decides which direction to readjust in using the data collected in array. Once the function decides the direction to readjust in, it calls that function.

```

void reAdjustWayBack(int direction);

```

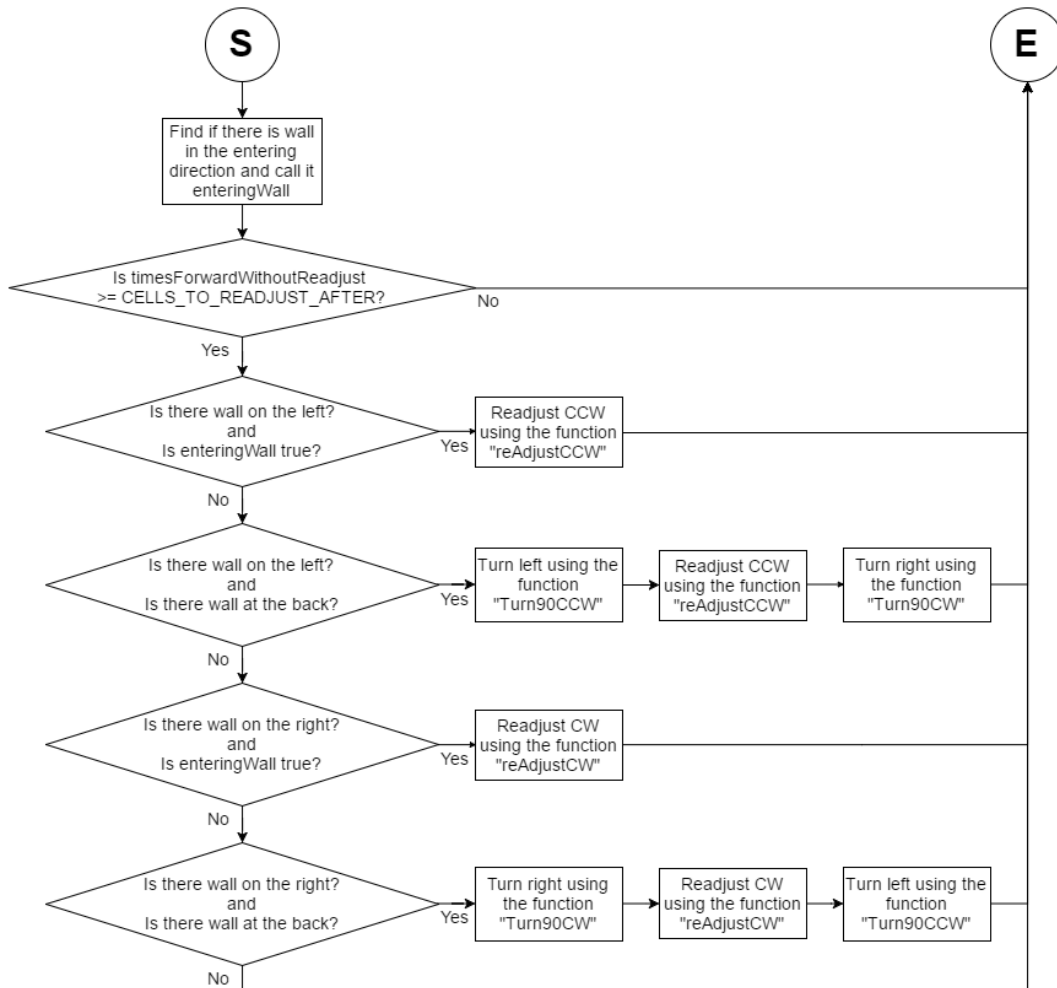


Figure 24: Flow Chart for Readjusting using walls detected

- The local variable direction is passed into the function but it does not return any variable
- Global variables and constants used are

```

timesForwardWithoutReadjust
CELLS_TO_READJUST_AFTER
PRESENT

```

- This function calls in other functions

```

thereIsWall();
reAdjustCW(int direction);
reAdjustCCW(int direction);
findLeft(int currentDirection);

```

```
findRight(int currentDirection);  
findBackDir(int currentDirection);  
isThereWallInDir(int wallDir);
```

3.15 Function for Movement All Together

- Function that implements the right following algorithm using the functions described above. Furthermore, it ensures that the robot readjusts whenever it can.

```
int MovementWithSensor(int direction);
```

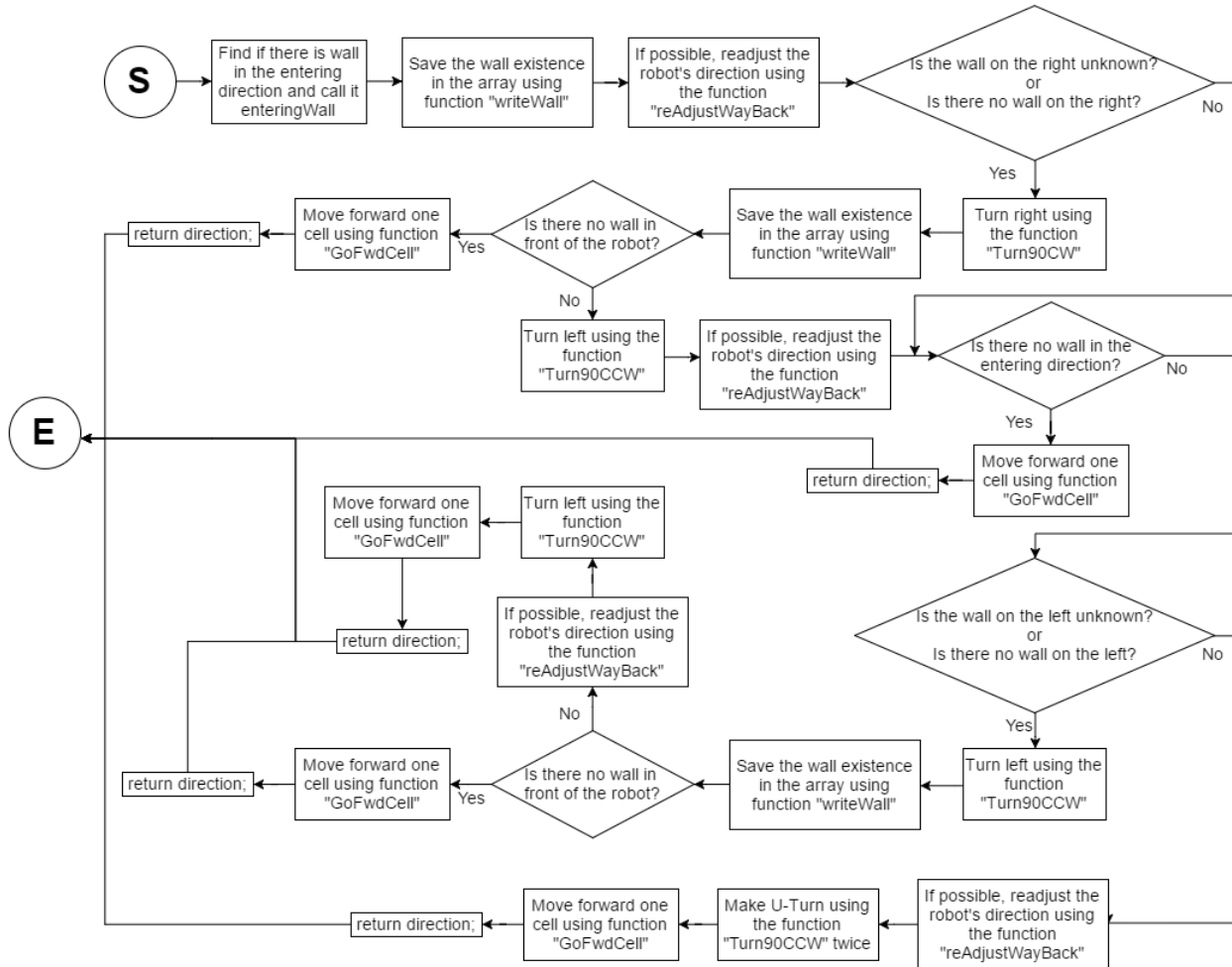


Figure 25: Flow Chart for Movement Function

- The local variable direction is passed into the function and it returns a new variable direction.
- Global variables and constants used are
UNKNOWN
NOT_PRESENT
- This function calls the other functions

```

writewall(int direction);
reAdjustWayBack(int direction);
isThereWallInDir(int wallDir);
findRight(int currentDirection);
thereIsWall();
goFwdCell(int direction);
Turn90CCW(int direction);
Turn90CW(int direction);

```


3.16 Functions for Returning Algorithm

- Function that deletes the duplicates from the array which saved up how the robot entered each cell. For example, if the robot moved two opposite directions in order, it is not necessary. Therefore, we delete the duplicates from the array

```
void deleteDuplicates();
```

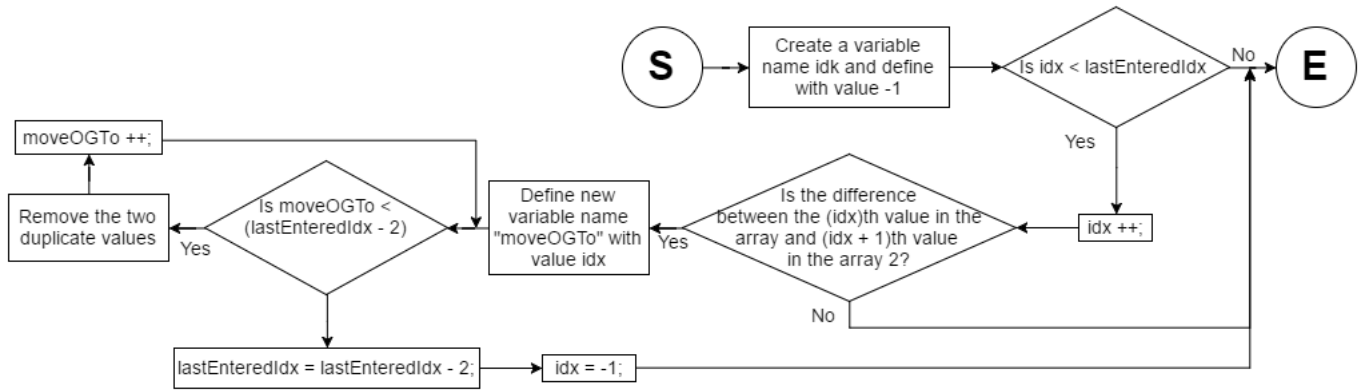


Figure 26: Flow Chart for Deleting Duplicate Function

Global variables used:

```
entered[]
lastEnteredIdx
```

- Function that reverses the direction from the array which saved up how the robot entered each cell. For example, if the robot went into the cell with direction East, then we change it to West. Therefore, we change all the directions to its opposite.

```
void reverseDirection();
```

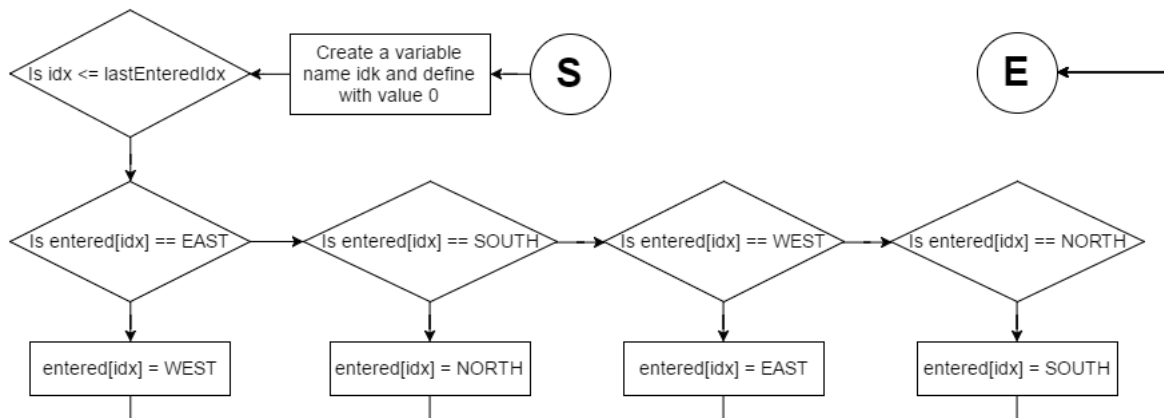


Figure 27: Flow Chart for Reversing Order of Values in the Array

Global variables used:

```
entered[]
lastEnteredIdx
```

- Function that takes in the variable direction and goes back to the initial position in the cell with the new array created by two functions above

```
void goingBackFastestRoute(int direction);
```

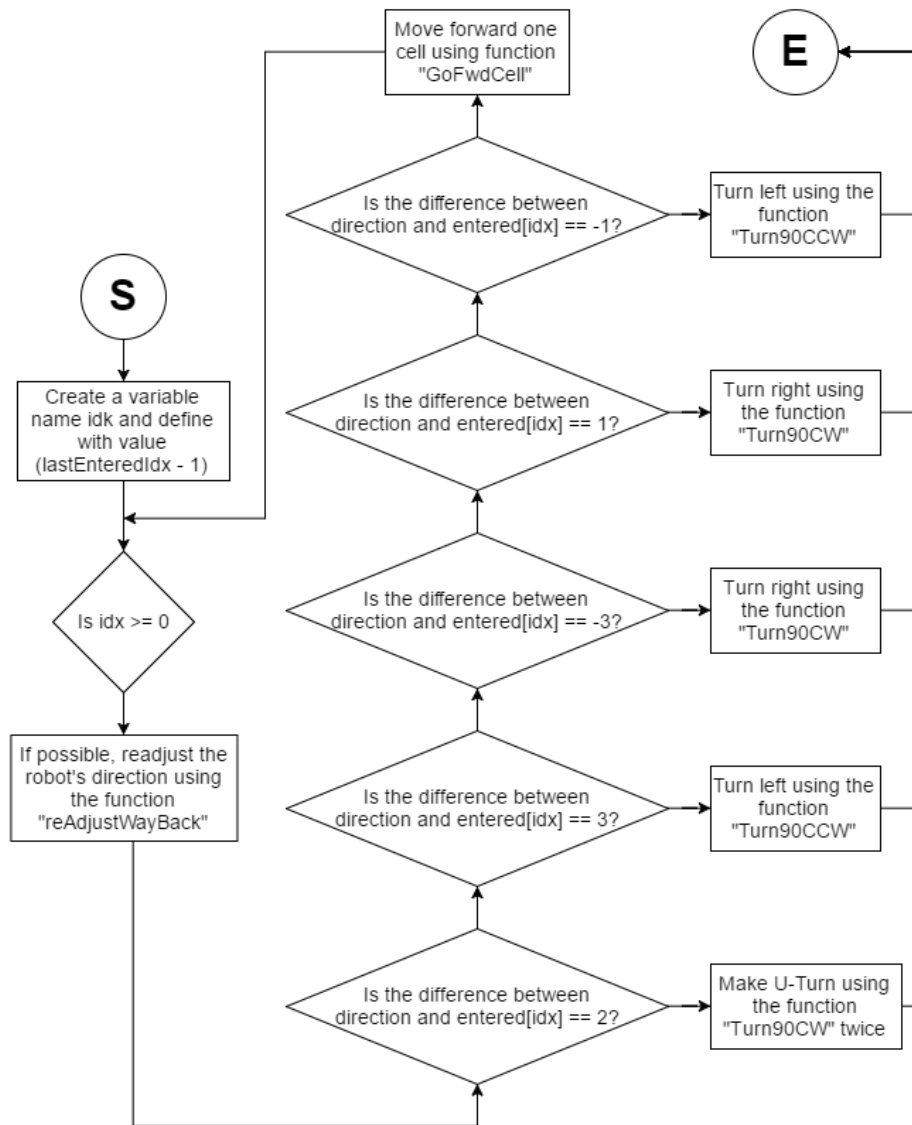


Figure 28: Flow Chart for Function to Go Back to Initial Position

- Global variables and constants used are

```
lastEnteredIdx
EAST
WEST
SOUTH
NORTH
entered[]
```

- This function calls in other functions

```
reAdjustWayBack(int direction);
```

```
Turn90CW(int direction);  
Turn90CCW(int direction);
```

3.17 Main Function

- More than ten functions were declared for simplicity of the main function. This function sums up all smaller functions

```
task main()
```

- A variable that represents current direction of the robot was declared. This is initialized as north as this is the orientation of the robot when it first enters the maze.

```
int direction = NORTH;
```

- Global variables and constants used are

```
MAZE_WIDTH
```

```
MAZE_HEIGHT
```

```
UNKNOWN
```

```
PRESENT
```

```
lastEnteredIdx
```

```
FREQUENCY
```

```
MILI_TO_BEEP_FOR
```

```
Maze[]
```

```
entered[]
```

- This function calls in other functions

```
MovementWithSensor(int direction);
```

```
deleteDuplicates();
```

```
reverseDirection();
```

```
goingBackFastestRoute(int direction);
```

```
drawInfo(int direction);
```

- Flow chart is on the next page

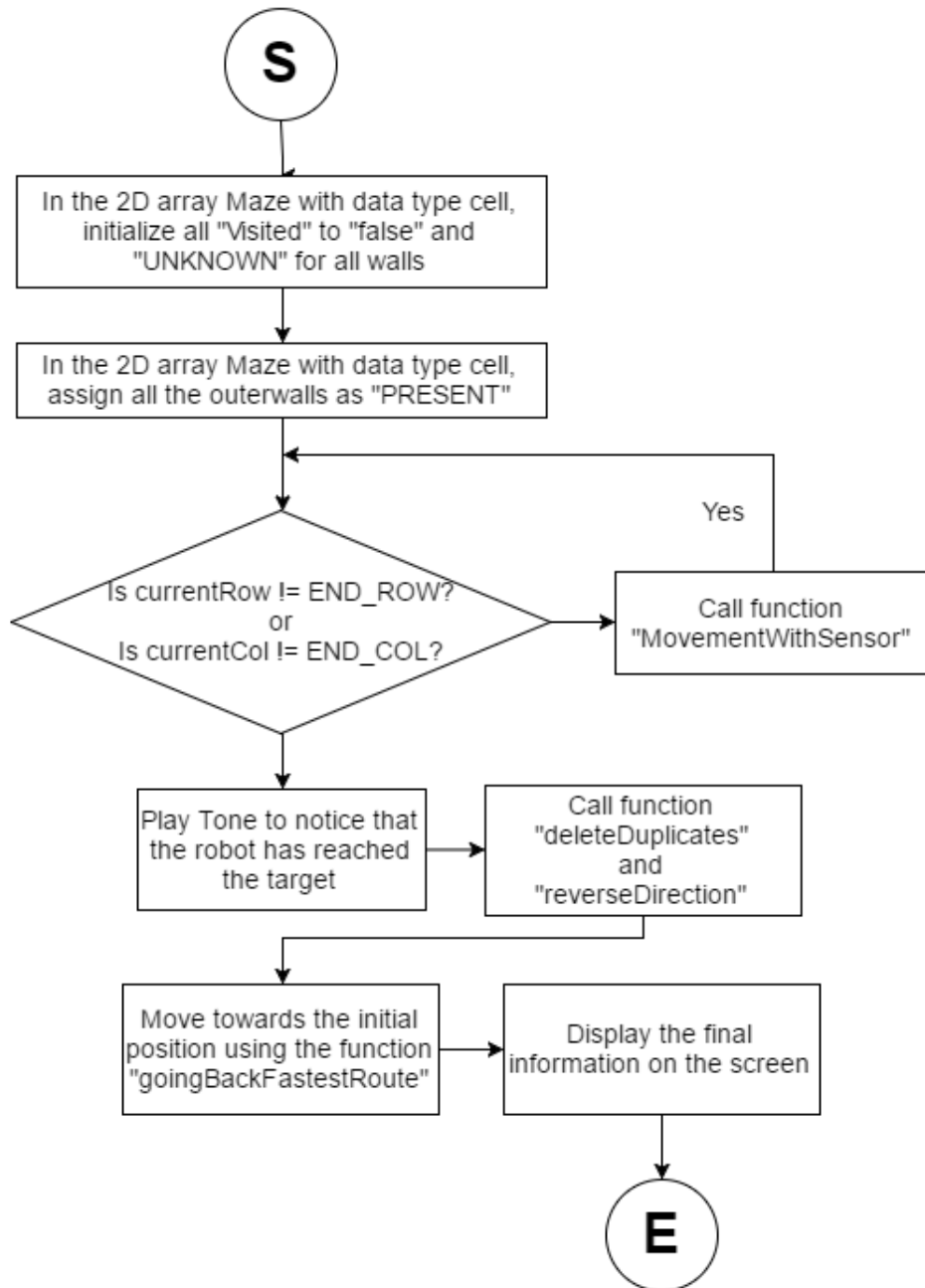


Figure 29: Flow Chart of the Main Function

4 Appendix

4.1 Full Source Code

```
1  #pragma config(Sensor, S1, distance, sensorEV3_Ultrasonic)
2  #pragma config(Motor, motorA, leftDrive, tmotorEV3_Large, PIDControl, driveLeft, encoder)
3  #pragma config(Motor, motorD, rightDrive, tmotorEV3_Large, PIDControl, driveRight, encoder)
4  /*!!Code automatically generated by 'ROBOTC' configuration wizard      !!*/
5
6  // Constants for robot's knowledge
7  #define NOT_PRESENT 0
8  #define PRESENT 1
9  #define UNKNOWN 2
10
11 // Maximum distance between robot and the wall
12 float const DIST_BETWEEN_BOT_AND_WALL = 7.6;
13
14 // Define directions using numbers
15 #define NORTH 0
16 #define EAST 1
17 #define SOUTH 2
18 #define WEST 3
19
20 typedef struct{
21     int NWall;
22     int SWall;
23     int EWall;
24     int WWall;
25     int Visited;
26     int entryDir;
27 }cell;
28
29 // Starting and End positions defined with Row and Column numbers
30 // These positions were used for the Demo
31 int const START_ROW = 2;
32 int const START_COL = 0;
33 int const END_ROW = 3;
34 int const END_COL = 4;
35 // (3,0) to (3,4) - longest route was the longest path for the practice
36
37 // Current position defined
38 int currentRow = START_ROW;
39 int currentCol = START_COL;
40
41 // Constants for beeping mechanism
42 int const MILI_TO_BEEP_FOR = 200;
43 int const FREQUENCY = 300;
44
45 // Uncertainty due to property of integer division of computer
46 float const UNCERTAINTY_STRAIGHT = 19;
47 float const UNCERTAINTY_ROT = 27;
```

```

48 float const UNCERTAINTY_READJUST = 35;
49
50 // Movement Variabels defined
51 float const ONE_ROTATION = 360 + UNCERTAINTY_STRAIGHT;
52 float const QUARTER_ROTATION = 180 + UNCERTAINTY_ROT;
53 float const SIZE_OF_ONE_CELL = 22.5425; //cm
54 float const DRIVE_GEAR_RATIO = 5;
55 float const DIAMETER_OF_WHEEL = 5.5; // cm
56 float const CIRCUMFERENCE_OF_WHEEL = PI * DIAMETER_OF_WHEEL;
57
58 // Speed Variable
59 int const FORWARD = -100;
60 int const BACKWARD = -FORWARD;
61
62 // MAZE VARIABLES
63 int const MAZE_WIDTH = 6;
64 int const MAZE_HEIGHT = 4;
65 int const LAST_MAZE_HEIGHT_INDEX = MAZE_HEIGHT - 1;
66 int const LAST_MAZE_WIDTH_INDEX = MAZE_WIDTH - 1;
67 cell Maze[MAZE_HEIGHT][MAZE_WIDTH];
68
69 // Array to save up how the robot entered each cells
70 int entered[MAZE_WIDTH*MAZE_HEIGHT*4];
71 int lastEnteredIdx = 0;
72
73 // Constants for displaying mechanism
74 #define SCREEN_HEIGHT 127
75 #define SCREEN_WIDTH 177
76 #define CELL_HEIGHT (SCREEN_HEIGHT / MAZE_HEIGHT)
77 #define CELL_WIDTH (SCREEN_HEIGHT / MAZE_WIDTH)
78 #define CELL_HEIGHT_MIDDLE (CELL_HEIGHT / 2)
79 #define CELL_WIDTH_MIDDLE (CELL_WIDTH / 2)
80
81 // Constants for readjusting mechanism
82 int const MILLISECS_TO_DRIVE_INTO_WALL = 1100;
83 int const CELLS_TO_READJUST_AFTER = 3;
84 int timesForwardWithoutReadjust = 0;
85
86 // Call functions
87 void goFwdCell(int direction);
88 int Turn90CW(int direction);
89 int Turn90CCW(int direction);
90 int MovementWithSensor(int direction);
91 void reverseDirection();
92 void deleteDuplicates();
93 int goingBackFastestRoute(int direction);
94 void drawInfo(int direction);
95 void reAdjustCCW(int direction);
96 void reAdjustCW(int direction);
97 int findLeft(int currentDirection);
98 int findRight(int currentDirection);

```

```

99     int findBackDir (int currentDirection);
100     int isThereWallInDir(int wallDir);
101     void reAdjustWayBack(int direction);
102
103
104     void drawInfo(int direction){
105         eraseDisplay();
106
107         for(int r = 0; r < MAZE_HEIGHT; r++){
108             for(int c = 0; c < MAZE_WIDTH; c++){
109
110                 if(Maze[r][c].SWall == PRESENT){
111                     drawLine(c*CELL_WIDTH,r*CELL_HEIGHT,c*CELL_WIDTH + CELL_WIDTH,r*CELL_HEIGHT);
112                 }
113                 if(Maze[r][c].NWall == PRESENT){
114                     drawLine(c*CELL_WIDTH,r*CELL_HEIGHT + CELL_HEIGHT,
115                             c*CELL_WIDTH + CELL_WIDTH,r*CELL_HEIGHT + CELL_HEIGHT);
116                 }
117                 if(Maze[r][c].WWall == PRESENT){
118                     drawLine(c*CELL_WIDTH,r*CELL_HEIGHT,c*CELL_WIDTH,
119                             r*CELL_HEIGHT + CELL_HEIGHT);
120                 }
121                 if(Maze[r][c].EWall == PRESENT){
122                     drawLine(c*CELL_WIDTH + CELL_WIDTH,r*CELL_HEIGHT,
123                             c*CELL_WIDTH + CELL_WIDTH, r*CELL_HEIGHT + CELL_HEIGHT);
124                 }
125             }
126         }
127
128
129         if(direction == NORTH){
130             displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
131                               currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "^");
132         }
133         else if(direction == EAST){
134             displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
135                               currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, ">");
136         }
137         else if(direction == WEST){
138             displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
139                               currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "<");
140         }
141         else if(direction == SOUTH){
142             displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
143                               currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "v");
144         }
145     }
146
147
148     task main(){
149

```



```

150     for (int c = 0; c < MAZE_WIDTH; c++){
151         for (int r = 0; r < MAZE_HEIGHT; r++){
152             Maze[r][c].Visited = false;
153             Maze[r][c].NWall = UNKNOWN;
154             Maze[r][c].SWall = UNKNOWN;
155             Maze[r][c].EWall = UNKNOWN;
156             Maze[r][c].WWall = UNKNOWN;
157         }
158     }
159
160     // Assigning walls [row][col]
161     for (int c = 0; c < MAZE_WIDTH; c++){
162         Maze[0][c].SWall = PRESENT;
163         Maze[LAST_MAZE_HEIGHT_INDEX][c].NWall = PRESENT;
164     }
165
166     for (int r = 0; r < MAZE_HEIGHT; r++){
167         Maze[r][0].WWall = PRESENT;
168         Maze[r][LAST_MAZE_WIDTH_INDEX].EWall = PRESENT;
169     }
170
171     int direction = NORTH;
172
173     Maze[currentRow][currentCol].entryDir = direction;
174     Maze[currentRow][currentCol].Visited = true;
175
176     while(currentRow != END_ROW || currentCol != END_COL){
177         direction = MovementWithSensor(direction);
178         entered[lastEnteredIdx] = direction;
179         lastEnteredIdx++;
180     }
181
182     playTone(FREQUENCY, MILI_TO_BEEP_FOR);
183
184
185     deleteDuplicates();
186
187     sleep(MILI_TO_BEEP_FOR * 10);
188
189     reverseDirection();
190     direction = goingBackFastestRoute(direction);
191
192     drawInfo(direction);
193
194     sleep(390000);
195 }
196
197 void deleteDuplicates(){
198     int idx = -1;
199
200

```

```

201     while(idx < lastEnteredIdx){
202         idx++;
203
204         if(abs(entered[idx] - entered[idx + 1]) == 2){
205             for(int moveOGTo = idx; moveOGTo <= lastEnteredIdx - 2; moveOGTo++){
206                 entered[moveOGTo] = entered[moveOGTo + 2];
207             }
208
209             lastEnteredIdx = lastEnteredIdx - 2;
210             idx = -1;
211         }
212     }
213 }
214
215
216 void reverseDirection(){
217     for(int idx = 0; idx <= lastEnteredIdx; idx++){
218         if(entered[idx]==EAST){
219             entered[idx] = WEST;
220         }
221         else if(entered[idx]==SOUTH){
222             entered[idx] = NORTH;
223         }
224         else if(entered[idx]==WEST){
225             entered[idx] = EAST;
226         }
227         else if(entered[idx]==NORTH){
228             entered[idx] = SOUTH;
229         }
230     }
231 }
232
233
234 int goingBackFastestRoute(int direction){
235
236     for(int idx = lastEnteredIdx - 1; idx >= 0; idx--){
237         reAdjustWayBack(direction);
238         int turnNum = entered[idx] - direction;
239
240         if(abs(turnNum) == 2){
241             direction = Turn90CW(direction);
242             direction = Turn90CW(direction);
243         }
244         else if(turnNum == 3){
245             direction = Turn90CCW(direction);
246         }
247         else if(turnNum == -3){
248             direction = Turn90CW(direction);
249         }
250         else if(turnNum == 1){
251             direction = Turn90CW(direction);

```

```

252     }
253     else if(turnNum == -1){
254         direction = Turn90CCW(direction);
255     }
256
257     goFwdCell(direction);
258 }
259 return direction;
260 }
261
262
263 void goFwdCell(int direction){
264     setMotorSyncEncoder(leftDrive, rightDrive, 0,
265                         (SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
266                         * ONE_ROTATION, FORWARD);
267
268     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
269
270     }
271
272     if (direction == NORTH){
273         Maze[currentRow][currentCol].NWall = false;
274         currentRow++;
275         Maze[currentRow][currentCol].SWall = false;
276     }
277     else if (direction == SOUTH){
278         Maze[currentRow][currentCol].SWall = false;
279         currentRow--;
280         Maze[currentRow][currentCol].NWall = false;
281     }
282     else if (direction == EAST){
283         Maze[currentRow][currentCol].EWall = false;
284         currentCol++;
285         Maze[currentRow][currentCol].WWall = false;
286     }
287     else if (direction == WEST){
288         Maze[currentRow][currentCol].WWall = false;
289         currentCol--;
290         Maze[currentRow][currentCol].EWall = false;
291     }
292
293     Maze[currentRow][currentCol].entryDir = direction;
294     Maze[currentRow][currentCol].Visited = true;
295
296     timesForwardWithoutReadjust++;
297 }
298
299
300 int Turn90CCW(int direction){
301     setMotorSyncEncoder(leftDrive, rightDrive, -100, QUARTER_ROTATION * DRIVE_GEAR_RATIO,
302                         FORWARD);

```

```

303         repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
304
305     }
306
307     direction = findLeft(direction);
308
309     drawInfo(direction);
310     return direction;
311 }
312
313
314
315 int Turn90CW(int direction){
316     setMotorSyncEncoder(leftDrive, rightDrive, 100, QUARTER_ROTATION * DRIVE_GEAR_RATIO,
317         FORWARD);
318
319     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
320
321     }
322
323     direction = findRight(direction);
324
325     drawInfo(direction);
326     return direction;
327 }
328
329
330 int thereIsWall(){
331     if(getUSDistance(distance)<=DIST_BETWEEN_BOT_AND_WALL || getUSDistance(distance)==255){
332         return 1;
333     }
334     return 0;
335 }
336
337
338 void writeWall(int direction){
339     if(direction == NORTH && thereIsWall()){
340         Maze[currentRow][currentCol].NWall = PRESENT;
341         if(currentRow + 1 <= LAST_MAZE_HEIGHT_INDEX){
342             Maze[currentRow + 1][currentCol].SWall = PRESENT;
343         }
344     }
345     else if(direction == SOUTH && thereIsWall()){
346         Maze[currentRow][currentCol].SWall = PRESENT;
347         if(currentRow - 1 >= 0){
348             Maze[currentRow - 1][currentCol].NWall = PRESENT;
349         }
350     }
351     else if(direction == EAST && thereIsWall()){
352         Maze[currentRow][currentCol].EWall = PRESENT;
353         if(currentCol + 1 <= LAST_MAZE_WIDTH_INDEX){

```

```

354         Maze[currentRow][currentCol + 1].WWall = PRESENT;
355     }
356 }
357 else if(direction == WEST && thereIsWall()){
358     Maze[currentRow][currentCol].WWall = PRESENT;
359     if(currentCol - 1 >= 0){
360         Maze[currentRow][currentCol - 1].EWall = PRESENT;
361     }
362 }
363 else if(direction == NORTH && !thereIsWall()){
364     Maze[currentRow][currentCol].NWall = false;
365     if(currentRow + 1 <= LAST_MAZE_HEIGHT_INDEX){
366         Maze[currentRow + 1][currentCol].SWall = false;
367     }
368 }
369 else if(direction == SOUTH && !thereIsWall()){
370     Maze[currentRow][currentCol].SWall = false;
371     if(currentRow - 1 >= 0){
372         Maze[currentRow - 1][currentCol].NWall = false;
373     }
374 }
375 else if(direction == EAST && !thereIsWall()){
376     Maze[currentRow][currentCol].EWall = false;
377     if(currentCol + 1 <= LAST_MAZE_WIDTH_INDEX){
378         Maze[currentRow][currentCol + 1].WWall = false;
379     }
380 }
381 else if(direction == WEST && !thereIsWall()){
382     Maze[currentRow][currentCol].WWall = false;
383     if(currentCol - 1 >= 0){
384         Maze[currentRow][currentCol - 1].EWall = false;
385     }
386 }
387 }
388
389
390 // Checking order, North(0), East(1), West(3) then South(2)
391 // right, north, left, back
392 int MovementWithSensor(int direction){
393
394     int enteringDirectionWall = thereIsWall();
395     writeWall(direction);
396     reAdjustWayBack(direction);
397
398     // turn to check if wall is right
399     if(isThereWallInDir(findRight(direction)) == UNKNOWN
400        || isThereWallInDir(findRight(direction)) == NOT_PRESENT){
401         direction = Turn90CW(direction);
402         writeWall(direction);
403
404         // go right if no wall right

```

```

405         if(!thereIsWall()){
406             goFwdCell(direction);
407             return direction;
408         }
409         else{
410             direction = Turn90CCW(direction);
411             reAdjustWayBack(direction);
412         }
413     }
414
415     if(!enteringDirectionWall){
416         goFwdCell(direction);
417         return direction;
418     }
419
420     // At this point, we know there r walls on the R and N
421     // We are facing N
422     // if we know there is wall left, go thru back
423     if(isThereWallInDir(findLeft(direction)) == UNKNOWN
424        || isThereWallInDir(findLeft(direction)) == NOT_PRESENT){
425
426         direction = Turn90CCW(direction);
427         writeWall(direction);
428
429         if(!thereIsWall()){
430             goFwdCell(direction);
431             return direction;
432         }
433         else{
434             reAdjustWayBack(direction);
435             direction = Turn90CCW(direction);
436             goFwdCell(direction);
437             return direction;
438         }
439     }
440     else{
441         reAdjustWayBack(direction);
442         direction = Turn90CCW(direction);
443         direction = Turn90CCW(direction);
444         goFwdCell(direction);
445         return direction;
446     }
447
448     sleep(1000000);
449 }
450
451 void reAdjustCCW(int direction){
452
453     direction = Turn90CCW(direction);
454
455

```

```

456     motor[rightDrive] = FORWARD;
457     motor[leftDrive] = FORWARD;
458     sleep(MILLISECS_TO_DRIVE_INTO_WALL);
459
460     setMotorSyncEncoder(leftDrive, rightDrive, 0, ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)
461         *DRIVE_GEAR_RATIO * ONE_ROTATION)/7 + UNCERTAINTY_READJUST,
462         BACKWARD);
463
464     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
465
466     }
467
468     Turn90CW(direction);
469
470     motor[rightDrive] = FORWARD;
471     motor[leftDrive] = FORWARD;
472     sleep(MILLISECS_TO_DRIVE_INTO_WALL);
473
474     setMotorSyncEncoder(leftDrive, rightDrive, 0,
475         ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
476         * ONE_ROTATION)/7 + UNCERTAINTY_READJUST, BACKWARD);
477
478     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
479
480     }
481
482     timesForwardWithoutReadjust = 0;
483 }
484
485 void reAdjustCW(int direction){
486
487     direction = Turn90CW(direction);
488
489     motor[rightDrive] = FORWARD;
490     motor[leftDrive] = FORWARD;
491     sleep(MILLISECS_TO_DRIVE_INTO_WALL);
492
493     setMotorSyncEncoder(leftDrive, rightDrive, 0,
494         ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
495         * ONE_ROTATION)/7 + UNCERTAINTY_READJUST, BACKWARD);
496
497     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
498
499     }
500
501     Turn90CCW(direction);
502
503     motor[rightDrive] = FORWARD;
504     motor[leftDrive] = FORWARD;
505     sleep(MILLISECS_TO_DRIVE_INTO_WALL);
506

```

```

507     setMotorSyncEncoder(leftDrive, rightDrive, 0,
508                          ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
509                          * ONE_ROTATION)/7 + UNCERTAINTY_READJUST, BACKWARD);
510
511     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
512
513     }
514
515     timesForwardWithoutReadjust = 0;
516 }
517
518
519 void reAdjustWayBack(int direction){
520     int enteringWall = thereIsWall();
521
522     if(timesForwardWithoutReadjust >= CELLS_TO_READJUST_AFTER){
523         if(isThereWallInDir(findLeft(direction)) == PRESENT && enteringWall){
524             reAdjustCCW(direction);
525         }
526         else if(isThereWallInDir(findLeft(direction)) == PRESENT &&
527                isThereWallInDir(findBackDir(direction)) == PRESENT){
528             direction = Turn90CCW(direction);
529             reAdjustCCW(direction);
530             direction = Turn90CW(direction);
531         }
532         else if(enteringWall && isThereWallInDir(findRight(direction)) == PRESENT){
533             reAdjustCW(direction);
534         }
535         else if(isThereWallInDir(findRight(direction)) == PRESENT &&
536                isThereWallInDir(findBackDir(direction)) == PRESENT){
537             direction = Turn90CW(direction);
538             reAdjustCW(direction);
539             direction = Turn90CCW(direction);
540         }
541     }
542 }
543
544
545 int findBackDir (int currentDirection){
546     if(currentDirection == NORTH){
547         return SOUTH;
548     }
549     else if(currentDirection == EAST){
550         return WEST;
551     }
552     else if(currentDirection == WEST){
553         return EAST;
554     }
555
556     return NORTH;
557 }

```



```

558
559
560 int findRight(int currentDirection){
561
562     if(currentDirection == WEST){
563         return NORTH;
564     }
565     else{
566         return currentDirection + 1;
567     }
568 }
569
570
571 int findLeft(int currentDirection){
572
573     if(currentDirection == NORTH){
574         return WEST;
575     }
576     else{
577         return currentDirection - 1;
578     }
579 }
580
581
582 int isThereWallInDir(int wallDir){
583     if(wallDir == NORTH && Maze[currentRow][currentCol].NWall == PRESENT){
584         return PRESENT;
585     }
586     else if(wallDir == SOUTH && Maze[currentRow][currentCol].SWall == PRESENT){
587         return PRESENT;
588     }
589     else if(wallDir == EAST && Maze[currentRow][currentCol].EWall == PRESENT){
590         return PRESENT;
591     }
592     else if(wallDir == WEST && Maze[currentRow][currentCol].WWall == PRESENT){
593         return PRESENT;
594     }
595
596     if(wallDir == NORTH && Maze[currentRow][currentCol].NWall == UNKNOWN){
597         return UNKNOWN;
598     }
599     else if(wallDir == SOUTH && Maze[currentRow][currentCol].SWall == UNKNOWN){
600         return UNKNOWN;
601     }
602     else if(wallDir == EAST && Maze[currentRow][currentCol].EWall == UNKNOWN){
603         return UNKNOWN;
604     }
605     else if(wallDir == WEST && Maze[currentRow][currentCol].WWall == UNKNOWN){
606         return UNKNOWN;
607     }
608

```

```
609         return NOT_PRESENT;
610     }
```