

Table of Contents

Contents

1	Progress Report	3
1.1	1 st Iteration	3
1.2	2 nd Iteration	3
1.3	3 rd Iteration	3
1.4	4 th Iteration	3
2	Mechanical Design of MazeBot	4
2.1	Top Level Mechanical Structure and Specifications	4
2.2	First Iteration	5
2.3	Second Iteration	7
3	Software Design of MazeBot	9
3.1	Constants and Variables Used to Define the Position of the Robot in the Maze and the Size of the Maze	10
3.2	Constants and Variables Used for Representation of Directions	11
3.3	Constants Used for Display	11
3.4	Constants Used for Moving Mechanism	12
3.5	Constants Used for Representation of Wall	13
3.6	Constants Used for Beeping Mechanism	13
3.7	Displaying Function	14
3.8	Moving Forward Function	15
3.9	Turning Functions	16
3.10	Wall Detecting Function	17
3.11	Function for Storing Data of the Walls	18
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	23
3.16	Functions for Returning Algorithm	24
3.17	Main Function	26
4	Appendix	28
4.1	Full Source Code	28
4.2	Link to Project Demonstration	40
4.3	Brickset Inventory	41

List of Figures

1	First Iteration Drive System	5
2	First Iteration Front View	5
3	First Iteration Side View	5
4	First Iteration Front View	5
5	First Iteration Side View	5
6	Second Iteration Drive System	7
7	Second Iteration Front View	7
8	Second Iteration Side View	7
9	Second Iteration Drive System	7
10	Second Iteration Front View	7
11	Second Iteration Side View	7
12	Proof of Degrees Formula	9
13	Flow Chart for Displaying Function	14
14	Flow Chart for Moving Forward	15
15	Flow Chart for Turning Right	16
16	Flow Chart for Turning Left	16
17	Flow Chart for Wall Detecting Function	17
18	Flow Chart for Storing Data Function	18
19	Flow Chart for Finding Back Function	19
20	Flow Chart for Finding Right Function	19
21	Flow Chart for Finding Left Function	19
22	Flow Chart for Finding Existence of Wall from the data	20
23	Flow Chart for Readjusting using Front wall and Right wall	21
24	Flow Chart for Readjusting using Back wall and Left wall	21
25	Flow Chart for Readjusting using walls detected	22
26	Flow Chart for Movement Function	23
27	Flow Chart for Deleting Duplicate Function	24
28	Flow Chart for Reversing Order of Values in the Array	24
29	Flow Chart for Function to Go Back to Initial Position	25
30	Flow Chart of the Main Function	27

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 a dependency for most of the other requirements. By tracking movement in the maze, 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 it has reached the target cell
- Robot stores wall information, visited/unvisited status and orientation at entry in a 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 the shortest path
Since we know the route we took to get to our final location, we can now implement the canceling algorithm (described in detail in **Subsection 3.16**).
- Robot displays current location graphically on screen
We are implementing this now, so that we could test our algorithm more easily. The last project feature we wanted to implement was to not check the same wall twice. This would be difficult to implement without sufficient debugging capabilities. As such, we improved our debugging capabilities before continuing.

1.4 4th Iteration

Should be achieved by: July 26, 2016

- Improve algorithm such that robot does not check same wall twice
Suppose the robot is in a cell with a wall to the South. If we've checked that there is a wall toward the South, the robot should store the fact that in the cell below our current cell, there is a wall to the North and act accordingly. In this case, the robot should not check North wall when it is in the cell below the current cell. The most general case will be programmed and described in **Section 3**. This has two major advantages: the robot does not waste time turning and the robot does not incur extra error because of unnecessary turning.

2 Mechanical Design of MazeBot

2.1 Top Level Mechanical Structure and Specifications

Our robot needed to have very accurate movement in order to be successful in the maze. This specification depends heavily on whether or not the motor encoders report accurate values to the algorithm. In order to have the motor encoders reporting accurately, 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 minimize mechanical error in movement. As a result, we will have to readjust after a certain number of cells (**Subsection 3.16**). However, having to readjust too often is problematic as this increases average time in each cell which is a major criteria point. As a result, we hope to minimize how often we need 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.

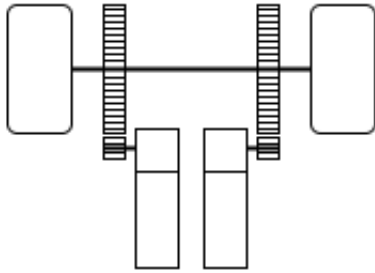


Figure 1: First Iteration Drive System

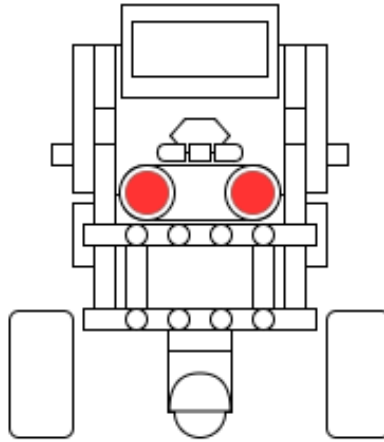


Figure 2: First Iteration Front View

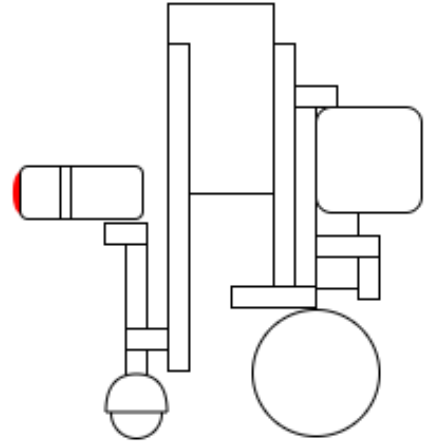


Figure 3: First Iteration Side View

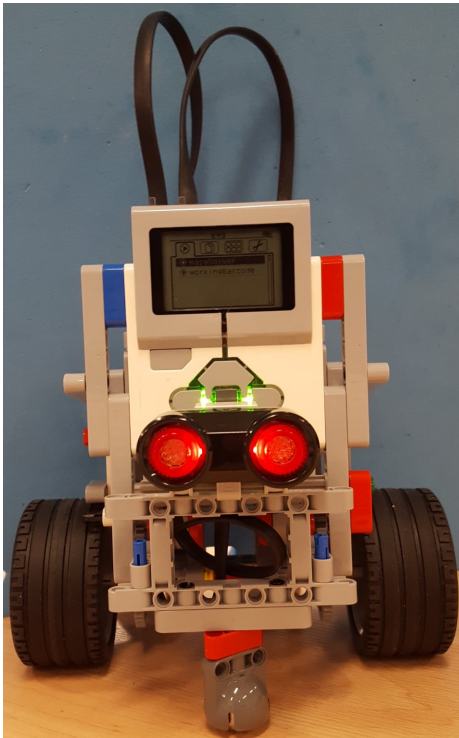


Figure 4: First Iteration Front View

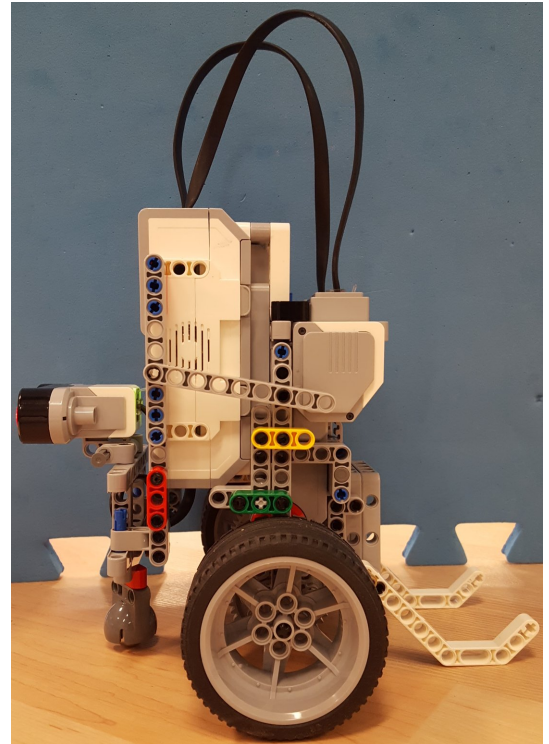


Figure 5: First Iteration Side View

- **Observations & Measurements**

- **Goal:** Robot is able to go 3 cells without needing to readjust
- **Failed:** Robot needed to readjust every second cell
- **Goal:** Robot is able to turn 90° accurately.
Test - An error of even $\pm 1^\circ$ will accumulate to a significant error when traversing cells. However, error in a single turn is hard to notice. Therefore, we chose to have the robot turn 90° 8 times in place in order to propagate any error significantly.
- **Failed:** Robot had error of $\pm 24^\circ$

- **Reasons for Test Failures**

- **Structural Integrity of the Drive System**

We were unable to find space to properly secure the left and right drive wheels. When testing, we found one wheel slipped forward and the other wheel slipped back when turning. This problem defeated the accuracy of the encoder. Hence, we were unable to meet our goal of accurate movement.

- **Robot is too Large.**

Since the brick is upright, the robot is top heavy. We needed two white bars in the back and one metal ball in the front to balance the robot. The additions of the two white bars and one metal ball negates the spacial advantage of having the robot's brick be upright. Even though the dimensions of the robot were within the size of one cell, it left very little room for movement error. As such, the robot began to run into walls after the second turn.

- **Wheels are too Big**

In order for the motor encoder to be accurate, the wheels must not slip. Therefore, to maximize friction, we decided to use the largest wheels in the brickset. 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 not have the brick upright. This will allow us to have enough room to properly secure the drive system. As such, the gears will make proper contact and will not 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. Similarly, we chose to use wheel with smaller radius but same thickness. Both of these changes will allow more room for movement 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.

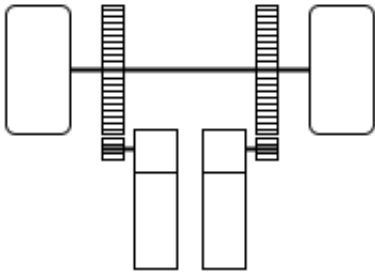


Figure 6: Second Iteration Drive System

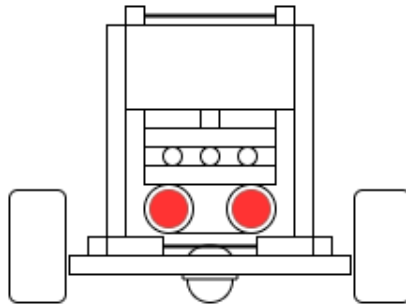


Figure 7: Second Iteration Front View

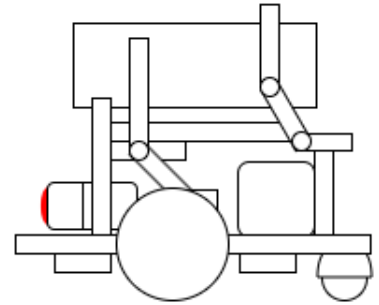


Figure 8: Second Iteration Side View

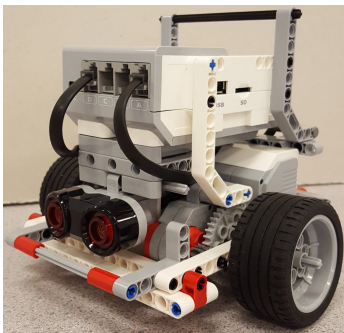


Figure 9: Second Iteration Drive System

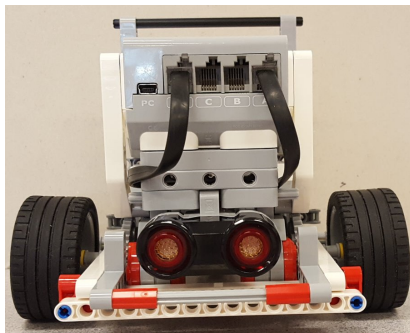


Figure 10: Second Iteration Front View

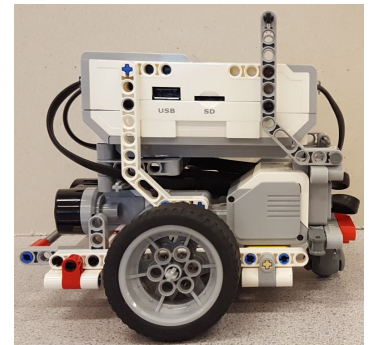


Figure 11: Second Iteration Side View

- **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 accumulate to a significant error when traversing cells. However, error in a single turn is hard to notice. Therefore, we chose to have the 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, the robot moved in a much more controlled way. Furthermore, having a more compact robot allowed the system to have a larger tolerance for movement error.

- **Conclusion**

After full 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 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 the same wall multiple times because turning is the robot's least accurate movement.

3 Software Design of MazeBot

To create a program that was suitable for requirements and easy to debug, we chose to employ a lot of functions. Each function does one small task and they were tested separately. These functions were then stitched together for different phases of the robot's maze solving algorithm.

These phases are:

Finding solution to the maze → Solving for the shortest path → Coming back the shortest route.

Furthermore, we wanted our program to have very few constants that we would need to test. For example, in order to move forward one cell, we needed to give the following function degrees to move each of our drive motors:

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

The degrees needed to move one cell forward could have been calculated by constantly testing different values of degrees. However, we chose to calculate the exact degrees that is needed for the robot to move exactly one cell forward. This approach in contrast to the former has two advantages:

1. It allows us to isolate any inaccurate movement problems as a purely 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:

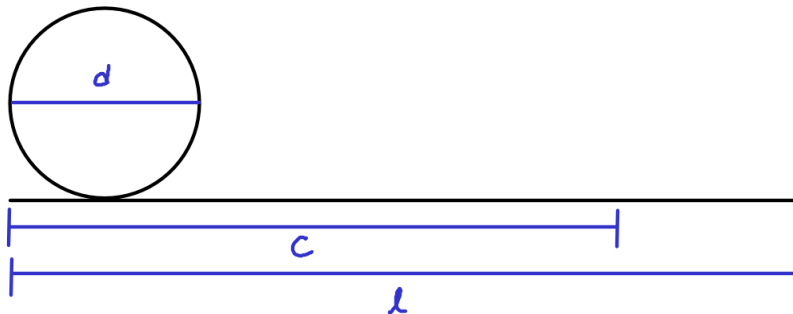


Figure 12: Proof of Degrees Formula

- d = diameter of the wheel
- $c = d \times \pi$ = circumference of the wheel
- l = distance of one cell
- 5 rotations of the gear attached to the motor rotates the wheel once
- The ratio between the distance of one cell and the circumference of the wheel is $\frac{l}{c}$

Therefore the degrees were calculated as:

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

3.1 Constants and 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 robot 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 a cell is four times in the 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 six parameters. For a given cell, this structure parameterized by where the walls are, whether the robot have visited the cell, and the direction that the robot entered 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 single cell's size on the screen were defined.

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

- Two constants that represent the robot's position in each cell in the screen were defined.

```
#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 three 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 Proportional Integral Derivative (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;
```

- Forward and backward 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 to readjust was defined. Timing algorithm was used because the flat surface at the front of the robot adjusts the robot 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 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 in the current cell.

```
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

- Two constants which represent the time and frequency of the beep when the robot finds 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. It also displays what it knows about the maze.

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

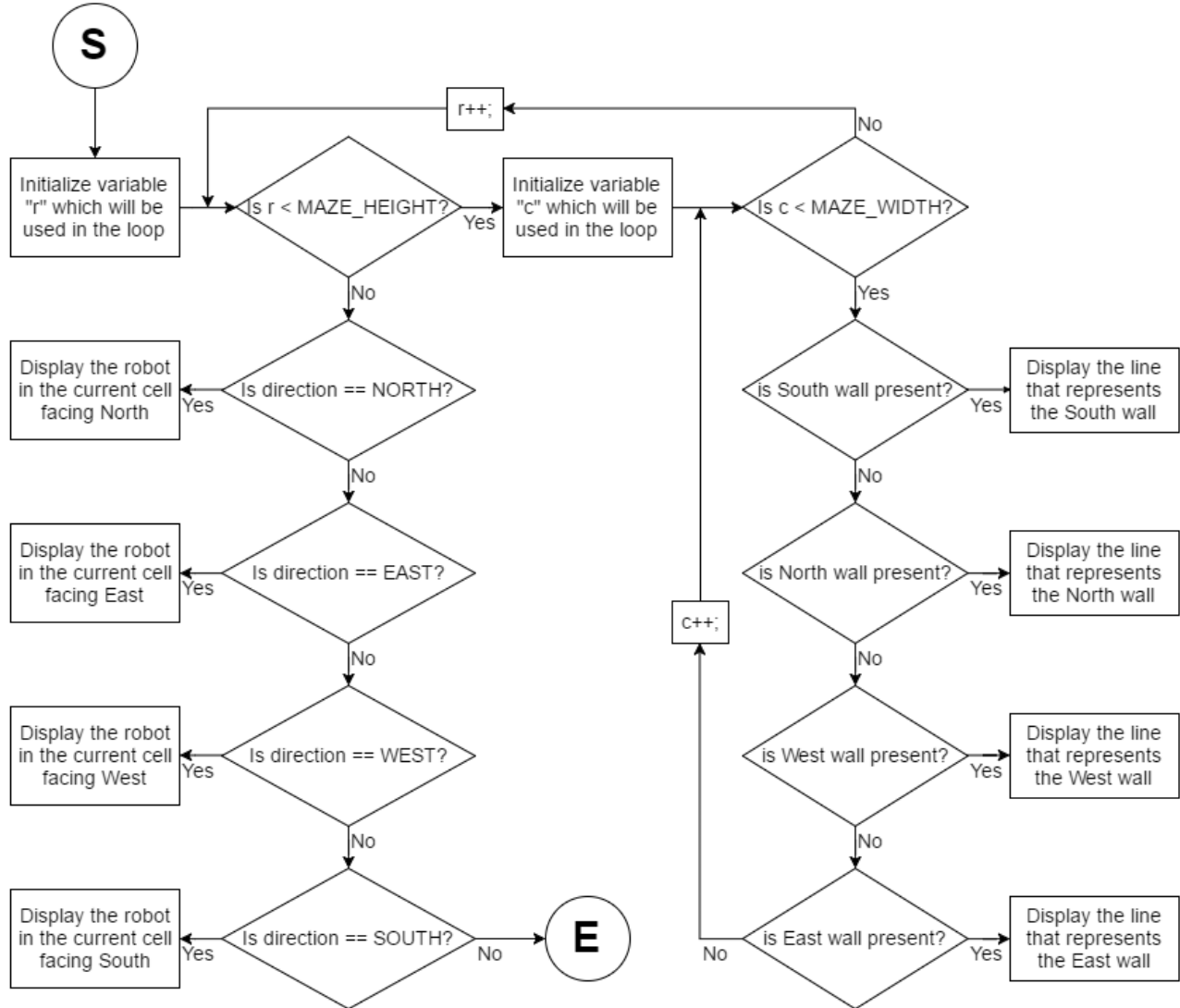


Figure 13: Flow Chart for Displaying Function

- 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[] [] .NWall  Maze[] [] .SWall  Maze[] [] .EWall  Maze[] [] .WWall
PRESENT
currentCol      currentRow
NORTH          SOUTH          EAST          WEST
  
```

3.8 Moving Forward Function

- This function moves the the robot forward exactly one cell forward. 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);
```

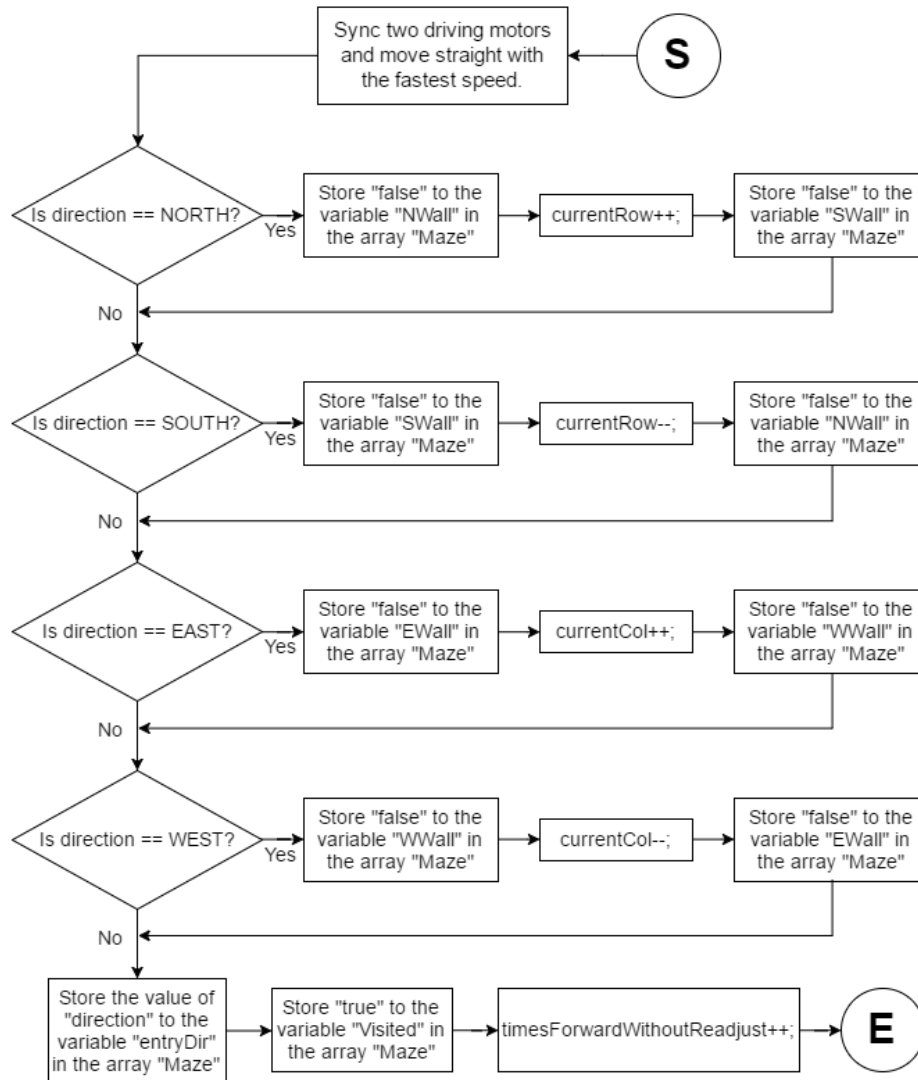


Figure 14: Flow Chart for Moving Forward

- 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
NORTH      SOUTH      EAST      WEST
currentCol      currentRow
Maze[] [] .NWall      Maze[] [] .SWall      Maze[] [] .EWall      Maze[] [] .WWall
timesForwardWithoutReadjust
  
```

3.9 Turning Functions

- Function for turning right 90°.

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

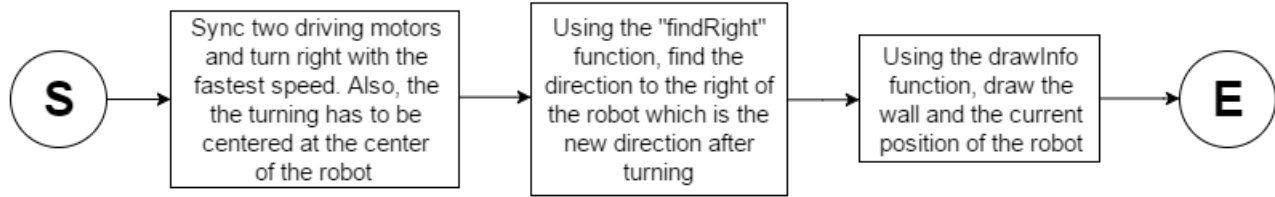


Figure 15: Flow Chart for Turning Right

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

```
QUARTER_ROTATION    DRIVE_GEAR_RATION  
FORWARD
```

- Functions called are:

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

- Function for turning left 90°.

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

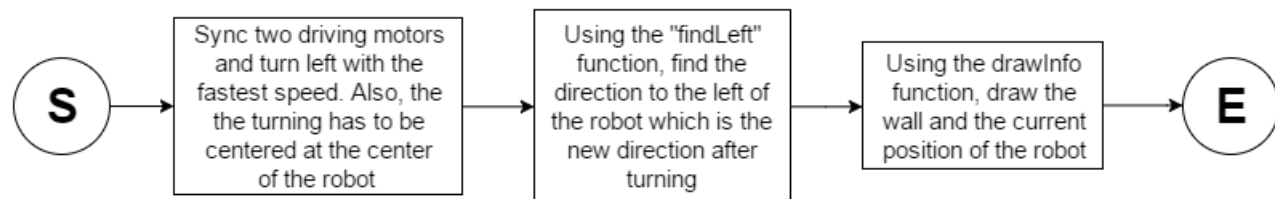


Figure 16: Flow Chart for Turning Left

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

```
QUARTER_ROTATION    DRIVE_GEAR_RATION  
FORWARD
```

- Functions called are:

```
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 robot.

```
int thereIsWall();
```

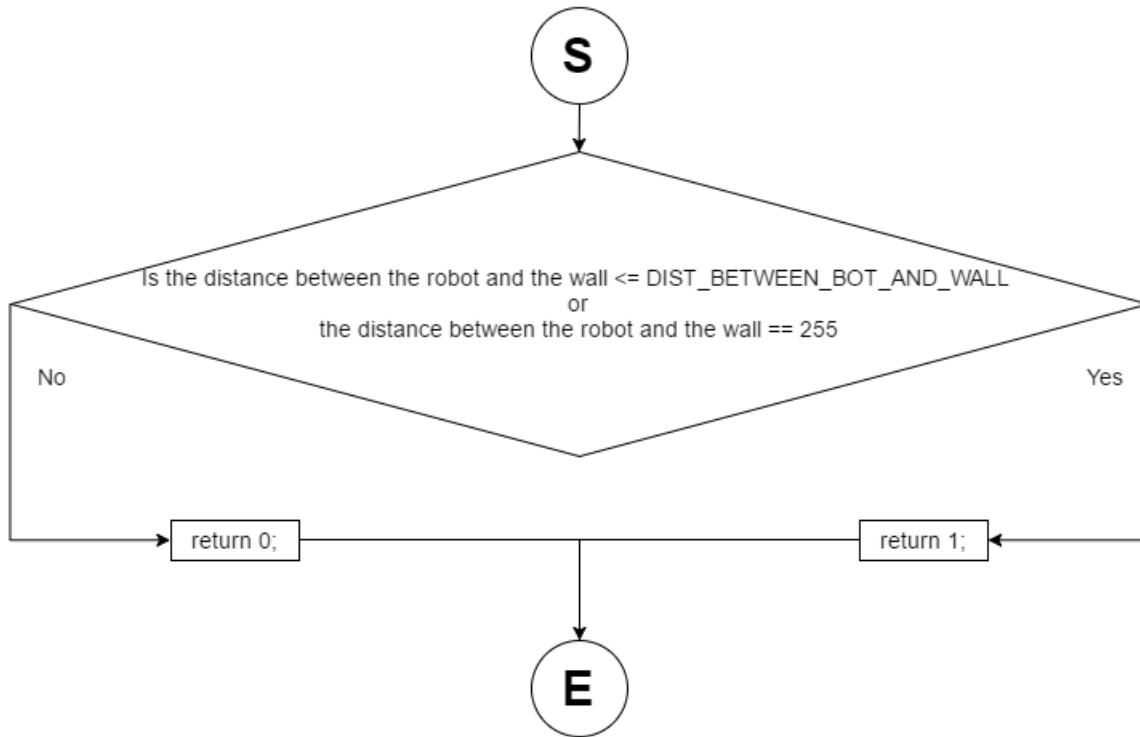


Figure 17: Flow Chart for Wall Detecting Function

- Very simple function that returns 1 if the sensor detects the wall.
- Global constant used is:
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 18: 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
currentCol    currentRow
Maze [] [] .NWall    Maze [] [] .SWall    Maze [] [] .EWall    Maze [] [] .WWall
PRESENT
LAST_MAZE_HEIGHT_INDEX    LAST_MAZE_WIDTH_INDEX
  
```

- Function called is:

```
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);
```

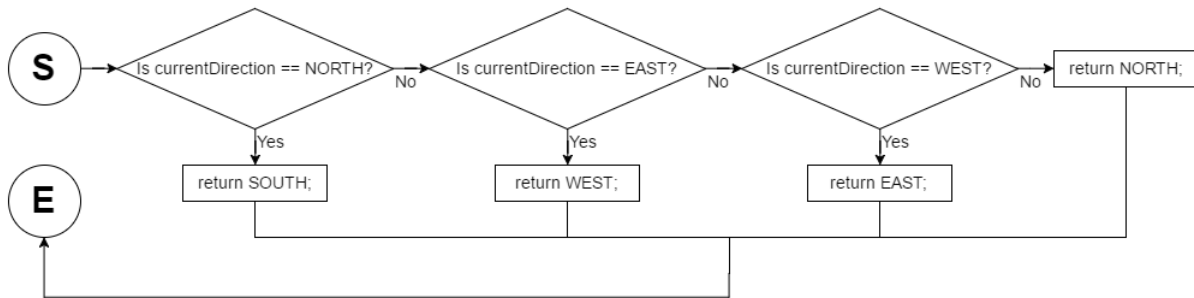


Figure 19: Flow Chart for Finding Back Function

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

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

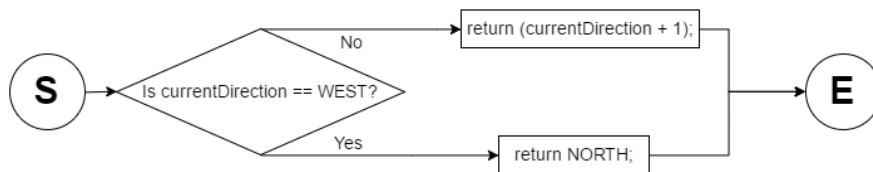


Figure 20: Flow Chart for Finding Right Function

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

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

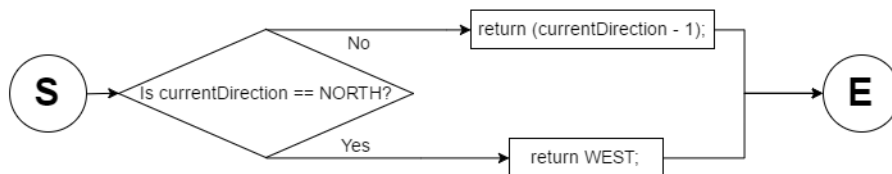


Figure 21: Flow Chart for Finding Left Function

– 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);
```

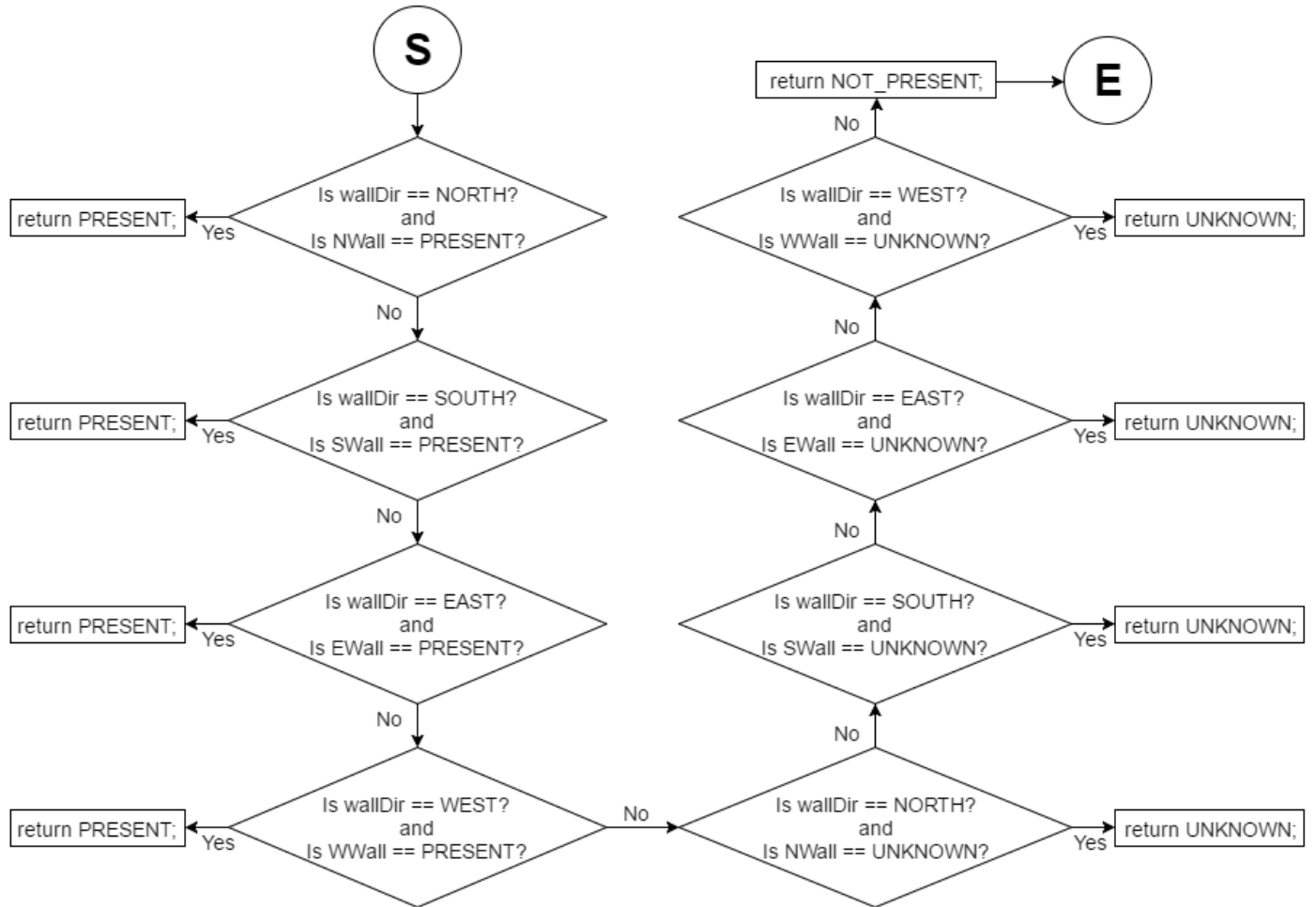


Figure 22: Flow Chart for Finding Existence of Wall from the data

– Global variables and constants used are:

NORTH	SOUTH	EAST	WEST
PRESENT	UNKNOWN	NOT_PRESENT	
Maze [] [] .NWall	Maze [] [] .SWall	Maze [] [] .EWall	Maze [] [] .WWall

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);
```

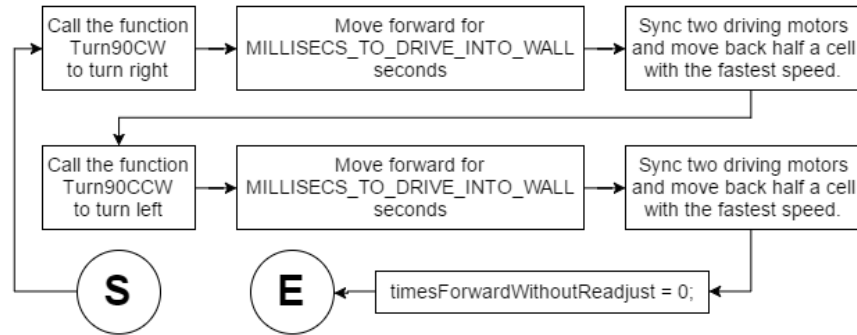


Figure 23: Flow Chart for Readjusting using Front wall and Right wall

- 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);
```

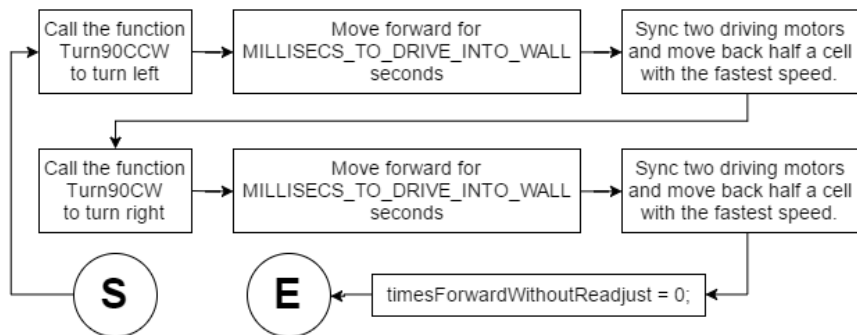


Figure 24: Flow Chart for Readjusting using Back wall and Left wall

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

```

SIZE_OF_ONE_CELL      CIRCUMFERENCE_OF_WHEEL
DRIVE_GEAR_RATIO      ONE_ROTATION      UNCERTAINTY_READJUST
FORWARD      BACKWARD      MILLISECS_TO_DRIVE_INTO_WALL
NORTH      SOUTH      EAST      WEST
Maze[] [] .NWall      Maze[] [] .SWall      Maze[] [] .EWall      Maze[] [] .WWall
timesForwardWithoutReadjust
  
```

- Functions called are:

```

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

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

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

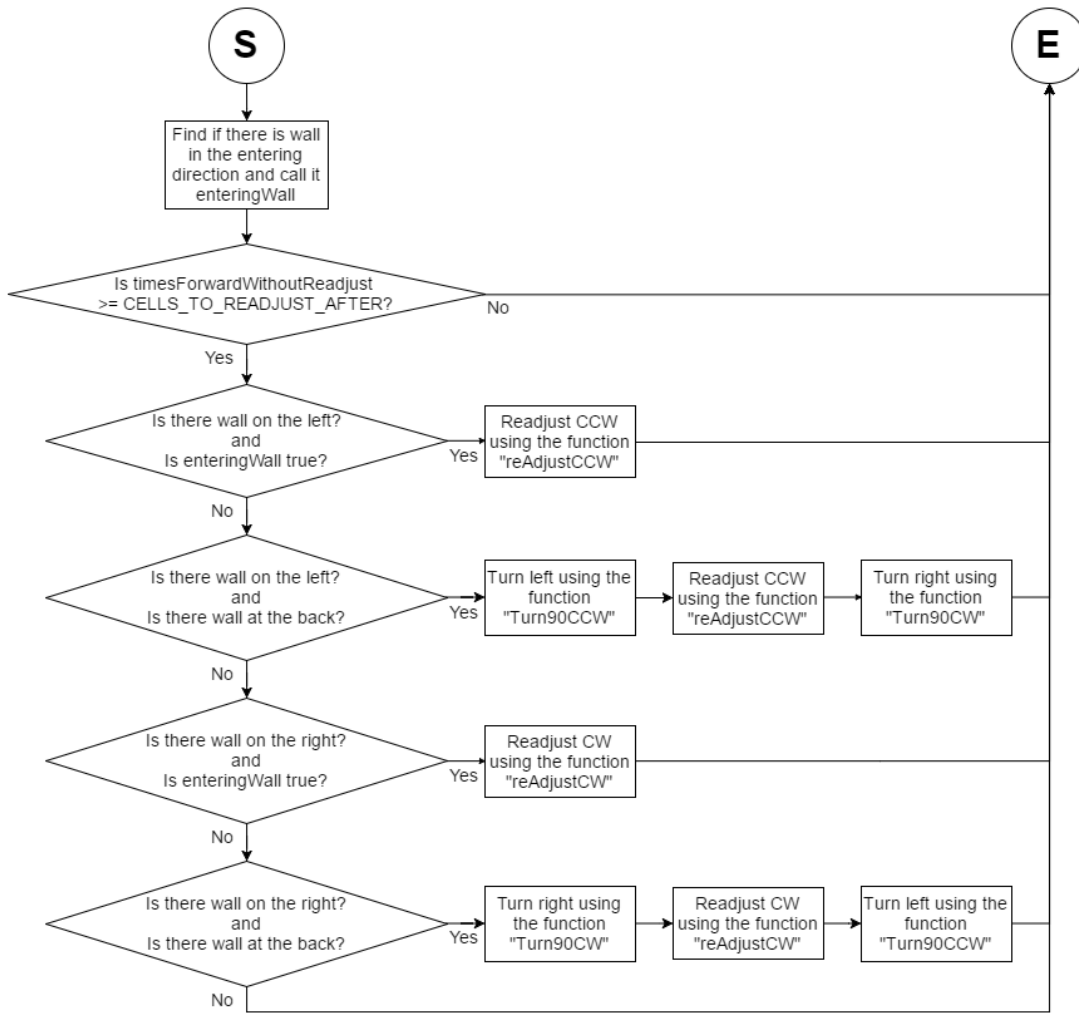


Figure 25: Flow Chart for Readjusting using walls detected

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

```
timesForwardWithoutReadjust
CELLS_TO_READJUST_AFTER
PRESENT
```

- Functions called are:

```
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);
```

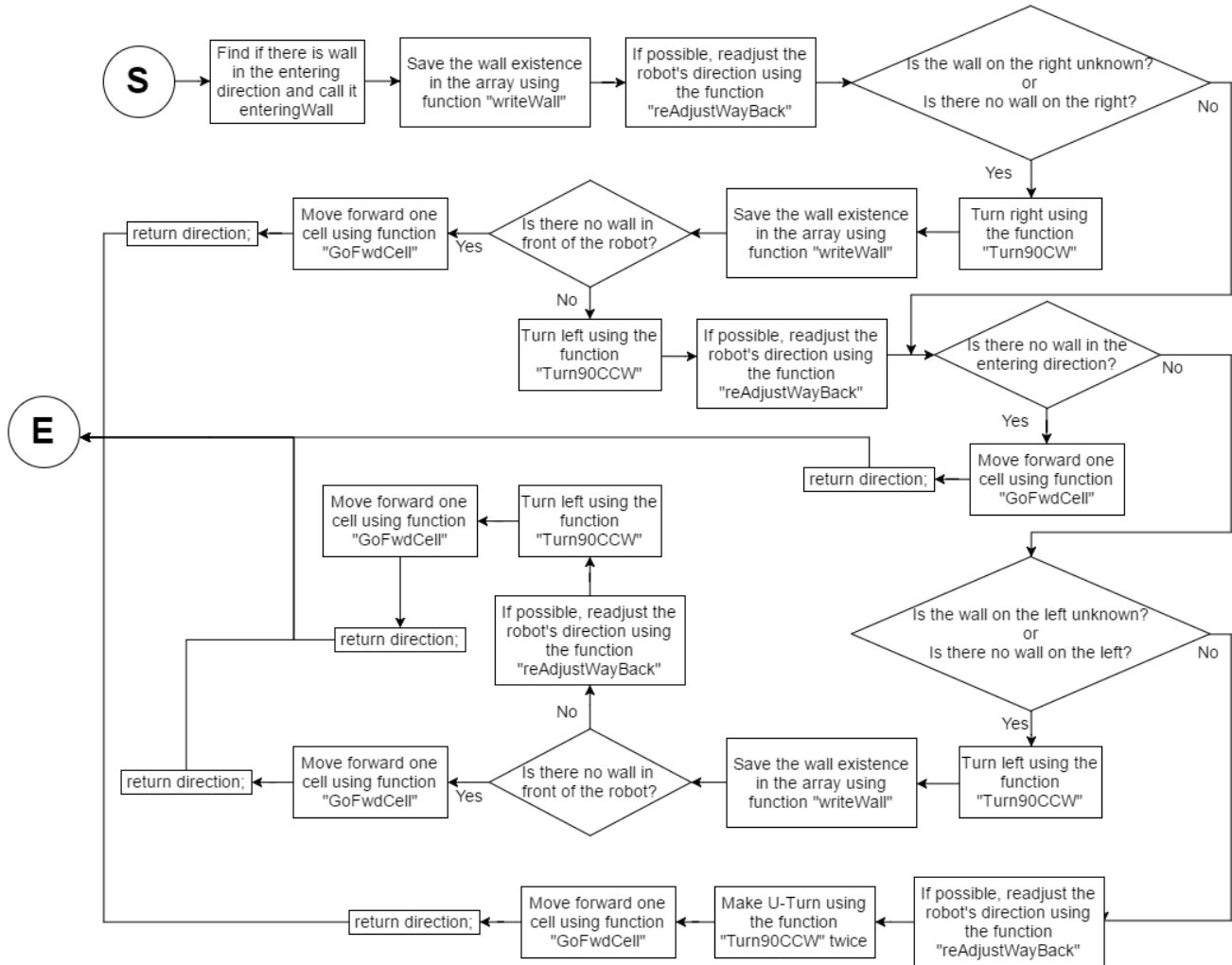


Figure 26: Flow Chart for Movement Function

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

NOT_PRESENT UNKNOWN

- Functions called are:

```

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 to follow those directions. Therefore, we delete the duplicates from the array.

```
void deleteDuplicates();
```

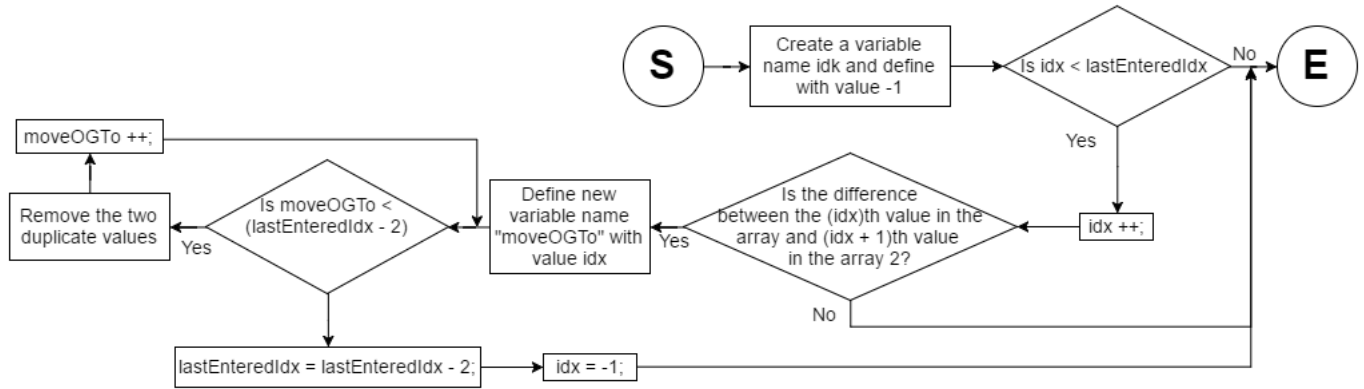


Figure 27: Flow Chart for Deleting Duplicate Function

- Global variables used are:

```
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();
```

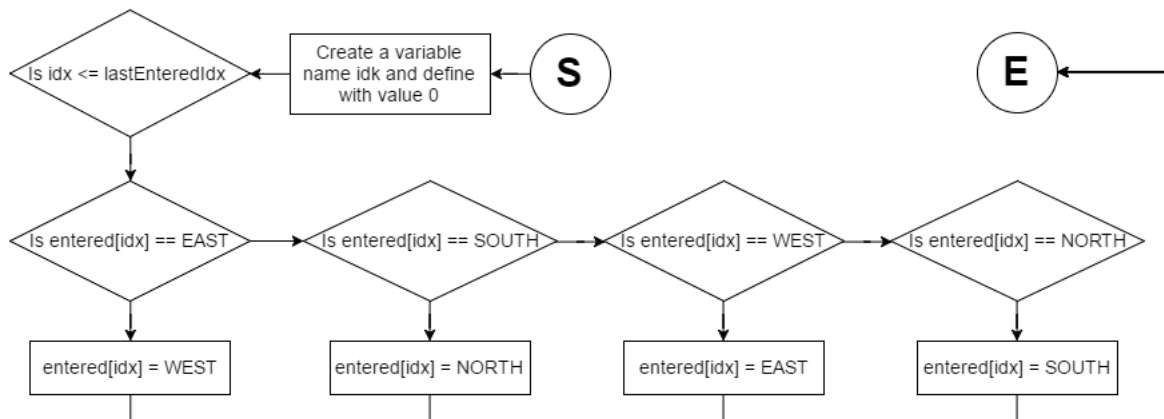


Figure 28: Flow Chart for Reversing Order of Values in the Array

- Global variables and constants used are:

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


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

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

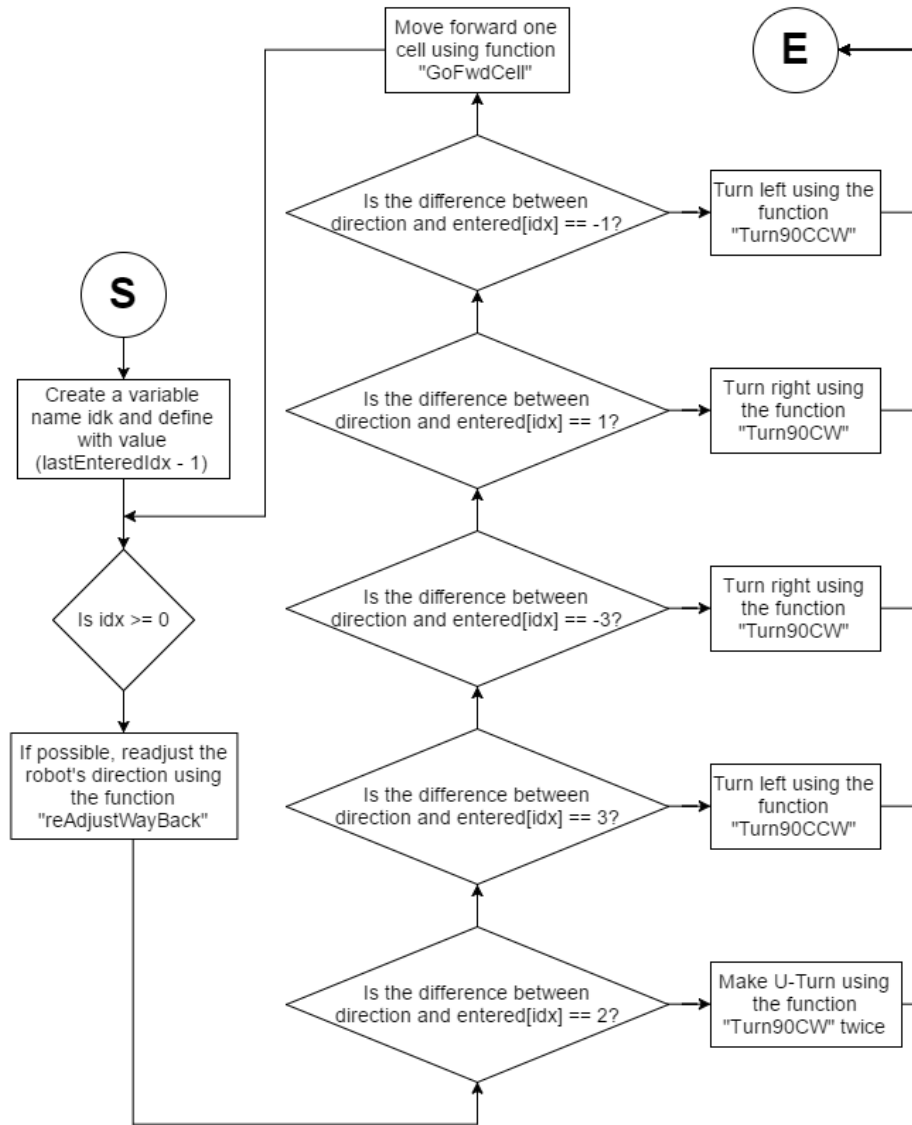


Figure 29: Flow Chart for Function to Go Back to Initial Position

- Global variables and constants used are:

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

- Functions called are:

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

3.17 Main Function

- More than ten functions were declared for simplicity of the main function. This function calls all the 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
entered[]       lastEnteredIdx
FREQUENCY       MILI_TO_BEEP_FOR
Maze[] [] .NWall   Maze[] [] .SWall   Maze[] [] .EWall   Maze[] [] .WWall
Maze[] [] .entryDir   Maze[] [] .Visited
PRESENT         UNKNOWN
LAST_MAZE_HEIGHT_INDEX   LAST_MAZE_WIDTH_INDEX
currentCol      currentRow
```

- Functions called are:

```
MovementWithSensor(int direction);
deleteDuplicates();
reverseDirection();
goingBackFastestRoute(int direction);
drawInfo(int direction);
```

- Flow chart is on the next page.

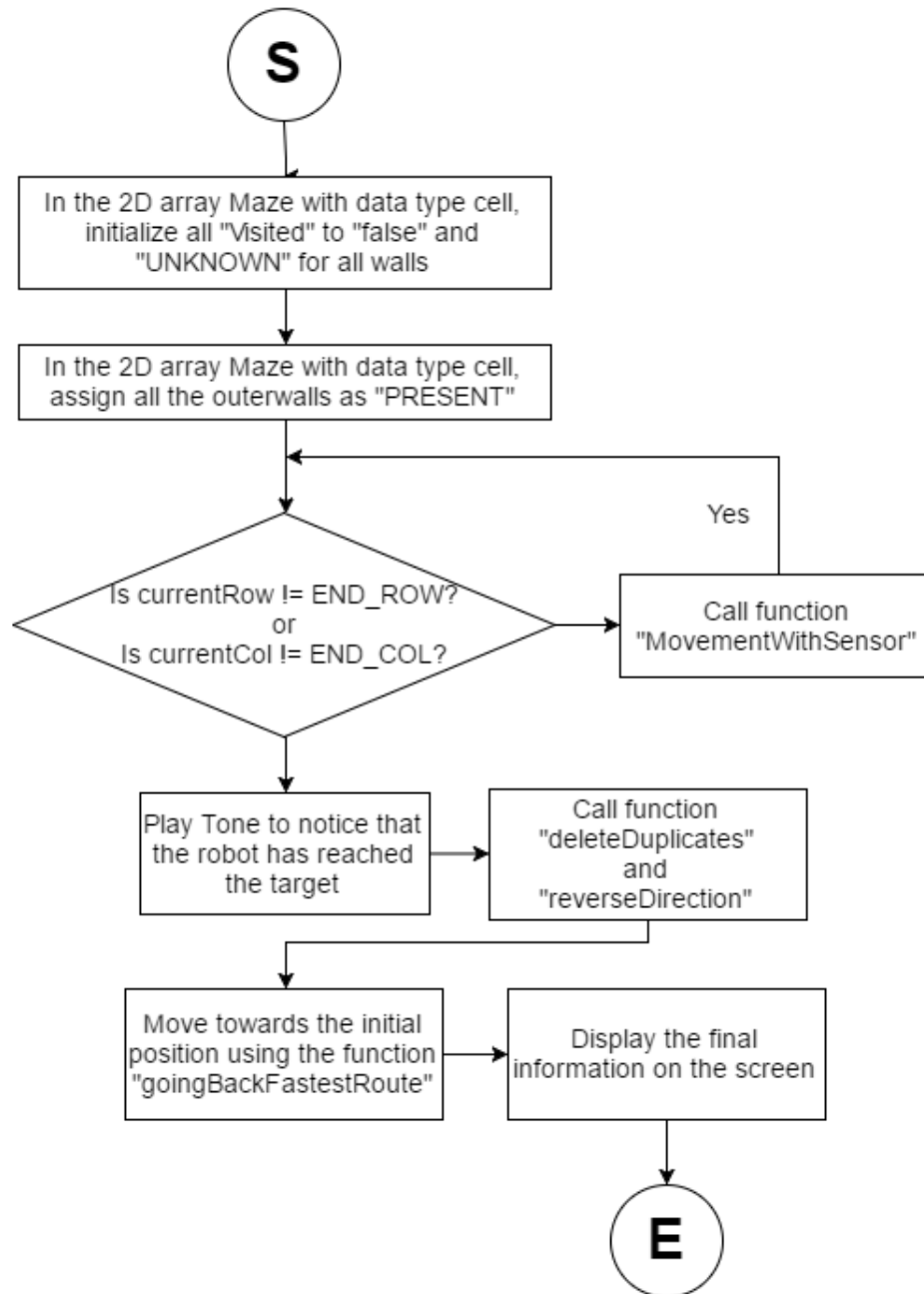


Figure 30: Flow Chart of the Main Function

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        if(direction == NORTH){
129            displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
130                currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "^");
131        }
132        else if(direction == EAST){
133            displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
134                currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, ">");
135        }
136        else if(direction == WEST){
137            displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
138                currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "<");
139        }
140        else if(direction == SOUTH){
141            displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
142                currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "v");
143        }
144    }
145
146
147
148
149

```

```

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

```

```

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

```



```

252         else if(turnNum == 1){
253             direction = Turn90CW(direction);
254         }
255         else if(turnNum == -1){
256             direction = Turn90CCW(direction);
257         }
258         goFwdCell(direction);
259     }
260     return direction;
261 }
262
263
264 void goFwdCell(int direction){
265     setMotorSyncEncoder(leftDrive, rightDrive, 0,
266                         (SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
267                         * ONE_ROTATION, FORWARD);
268
269     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
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
301
302

```

```

303 int Turn90CCW(int direction){
304     setMotorSyncEncoder(leftDrive, rightDrive, -100, QUARTER_ROTATION * DRIVE_GEAR_RATIO,
305                         FORWARD);
306
307     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
308
309     }
310     direction = findLeft(direction);
311     drawInfo(direction);
312     return direction;
313 }
314
315
316 int Turn90CW(int direction){
317     setMotorSyncEncoder(leftDrive, rightDrive, 100, QUARTER_ROTATION * DRIVE_GEAR_RATIO,
318                         FORWARD);
319
320     repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){
321
322     }
323     direction = findRight(direction);
324     drawInfo(direction);
325     return direction;
326 }
327
328
329 int thereIsWall(){
330     if(getUSDistance(distance)<=DIST_BETWEEN_BOT_AND_WALL || getUSDistance(distance)==255){
331         return 1;
332     }
333     return 0;
334 }
335
336
337 void writeWall(int direction){
338     if(direction == NORTH && thereIsWall()){
339         Maze[currentRow][currentCol].NWall = PRESENT;
340         if(currentRow + 1 <= LAST_MAZE_HEIGHT_INDEX){
341             Maze[currentRow + 1][currentCol].SWall = PRESENT;
342         }
343     }
344     else if(direction == SOUTH && thereIsWall()){
345         Maze[currentRow][currentCol].SWall = PRESENT;
346         if(currentRow - 1 >= 0){
347             Maze[currentRow - 1][currentCol].NWall = PRESENT;
348         }
349     }
350     else if(direction == EAST && thereIsWall()){
351         Maze[currentRow][currentCol].EWall = PRESENT;
352         if(currentCol + 1 <= LAST_MAZE_WIDTH_INDEX){
353             Maze[currentRow][currentCol + 1].WWall = PRESENT;

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

4.2 Link to Project Demonstration

For project demonstration, watch [Video Presentation](#) or go to the next url: <https://drive.google.com/open?id=0B89f8wQs4pbuYUs4OFhBX3ZaUnM>

4.3 Brickset Inventory

This is done separately from the Report.

There were some extra parts in the locker, but we did not added them to our inventory because the pieces were not originally in the brickset.

We found the extra pieces in the randomly assigned locker (Locker number 13).

Following pages show the Brickset Inventory.