Table of Contents

Contents

List of Figures

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 accurrately, 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.

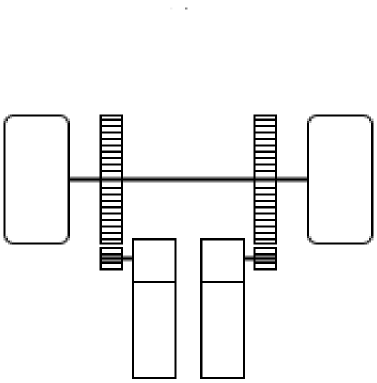

Figure 1: First Iteration Drive System

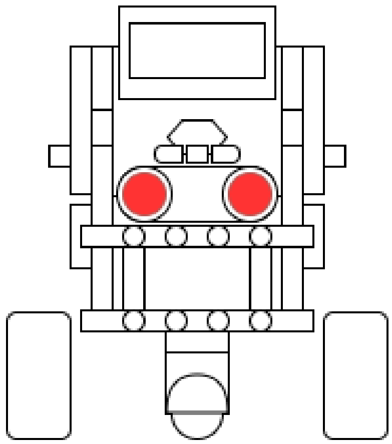

Figure 2: First Iteration Front View

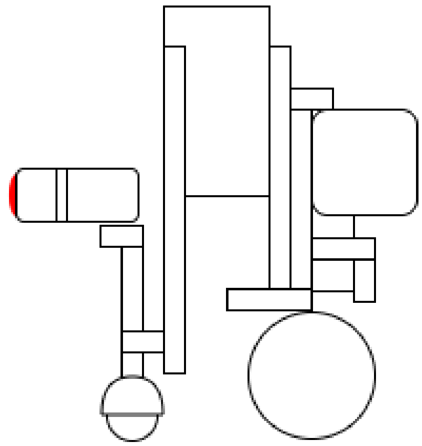

Figure 3: First Iteration Side View

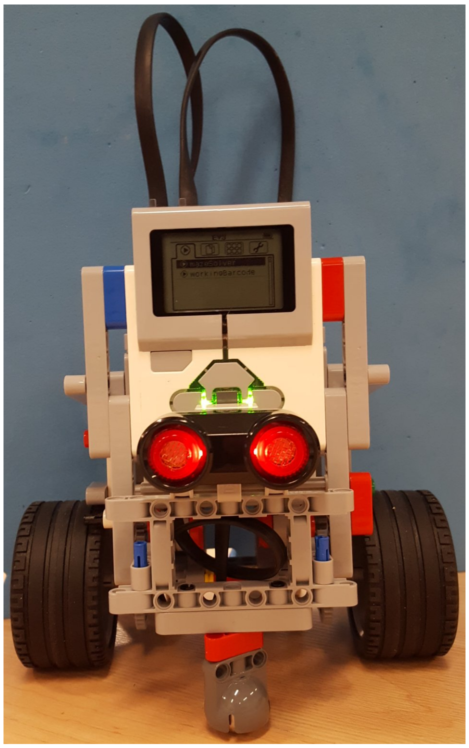

Figure 4: First Iteration Front View

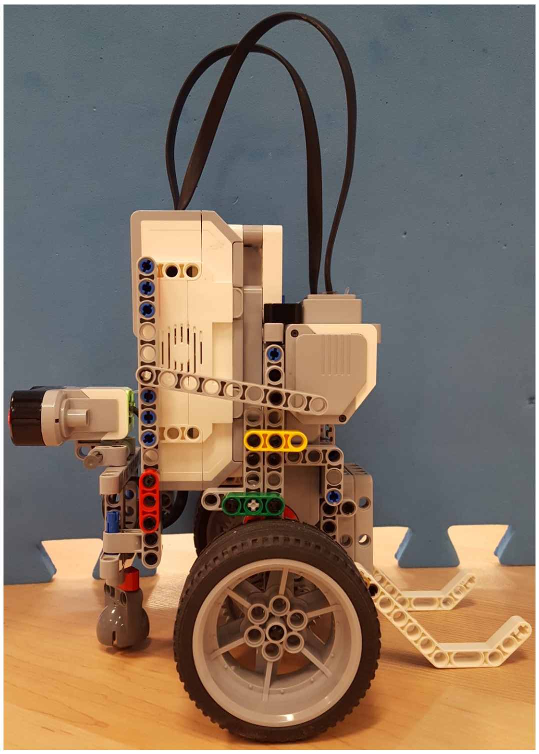

Figure 5: First Iteration Side View

- **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 $\pm 1°$ 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 $\pm 24°$
- **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. Similarily, 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.

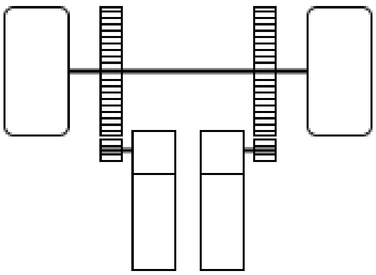

Figure 6: Second Iteration Drive System

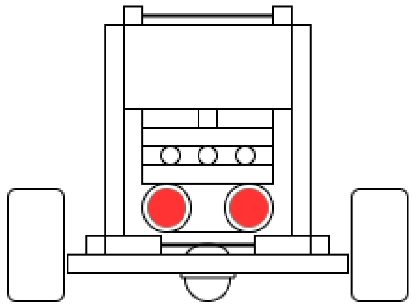

Figure 7: Second Iteration Front View

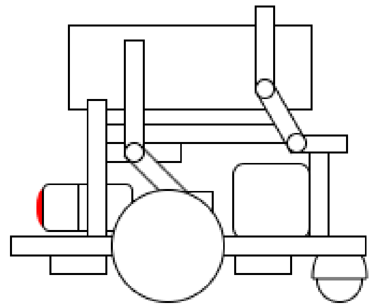

Figure 8: Second Iteration Side View

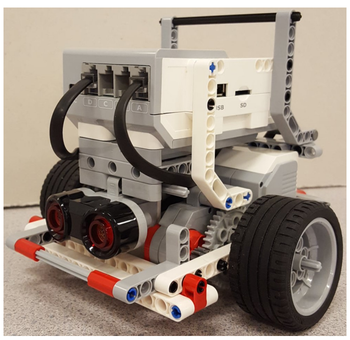

Figure 9: Second Iteration Drive System

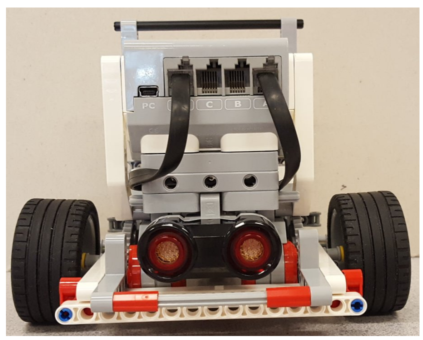

Figure 10: Second Iteration Front View

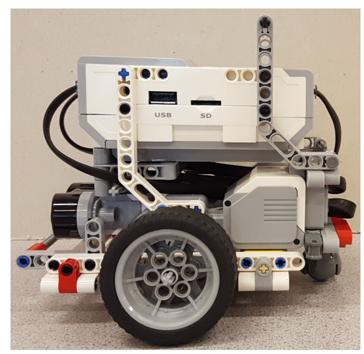

Figure 11: Second Iteration Side View

- **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 ±1° 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:

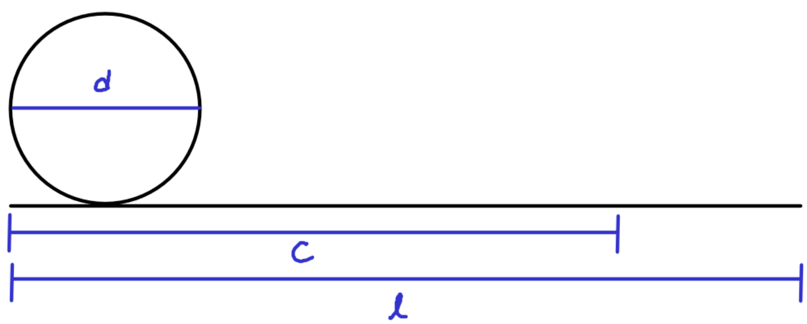

Figure 12: Proof of Degrees Formula

- 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 MILISECS_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 MILI_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);
  ```

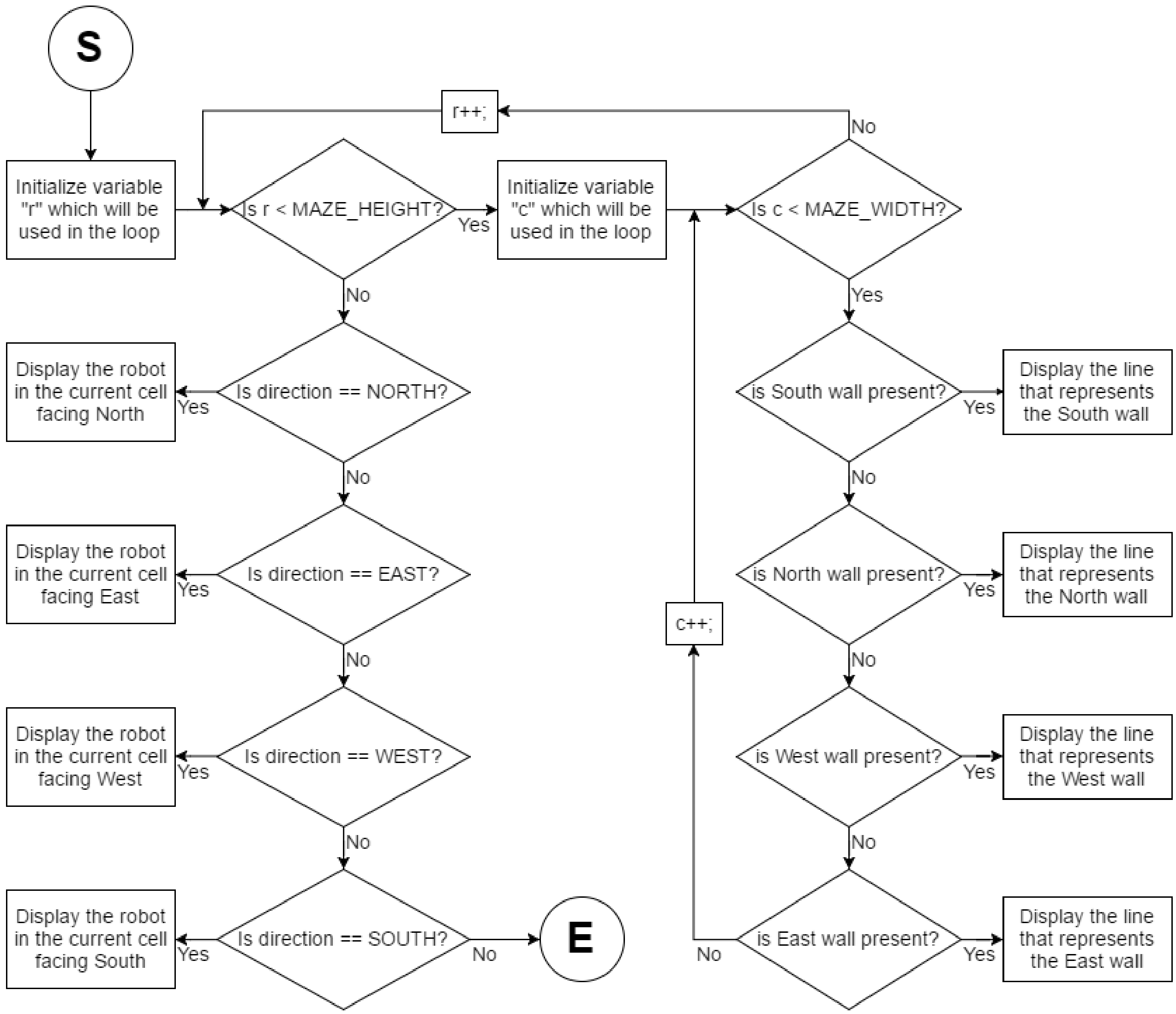

Figure 13: Flow Chart for Displaying Function

  - 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);
```

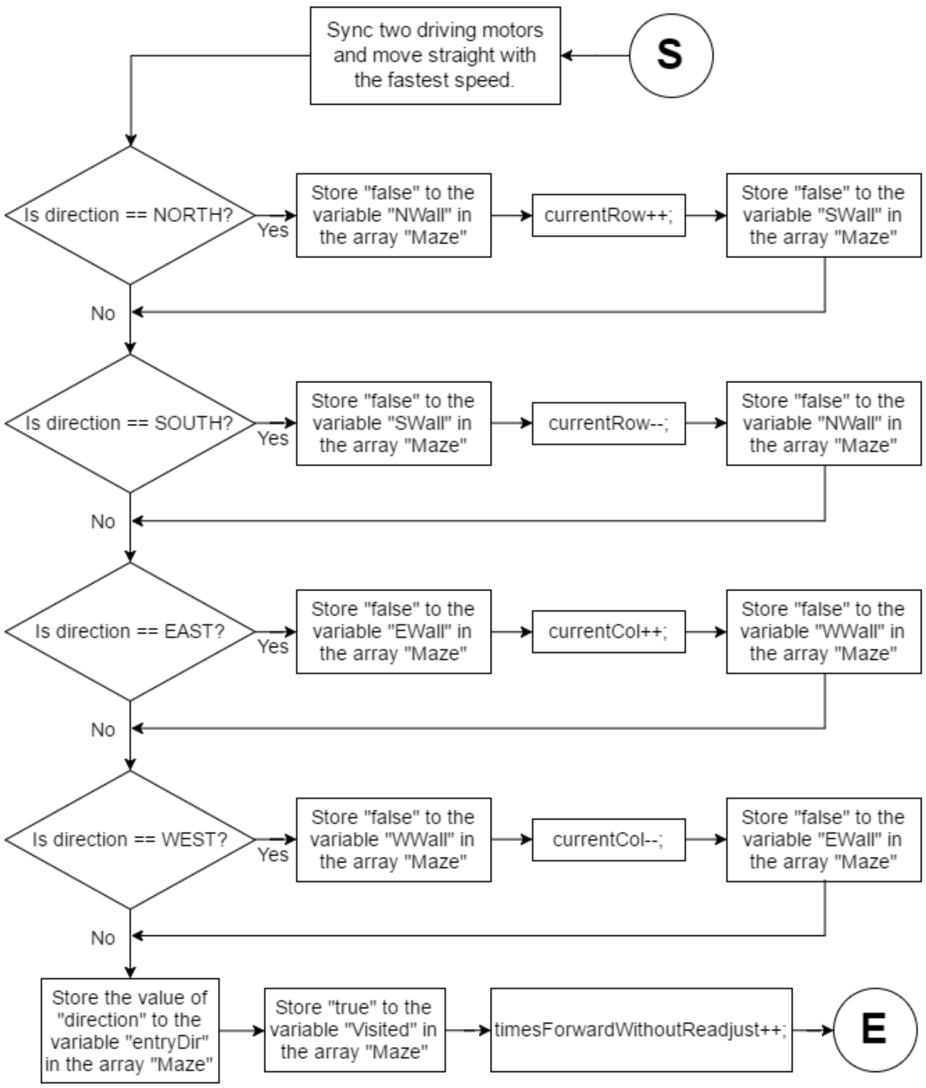

Figure 14: Flow Chart for Moving Forward

- 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);
```

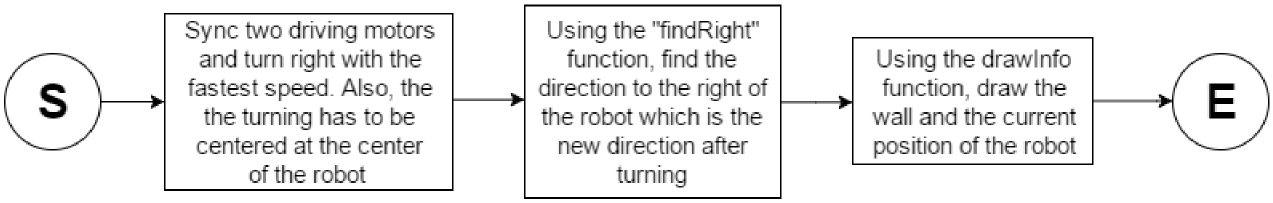

Figure 15: Flow Chart for Turning Right

  - 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);
```

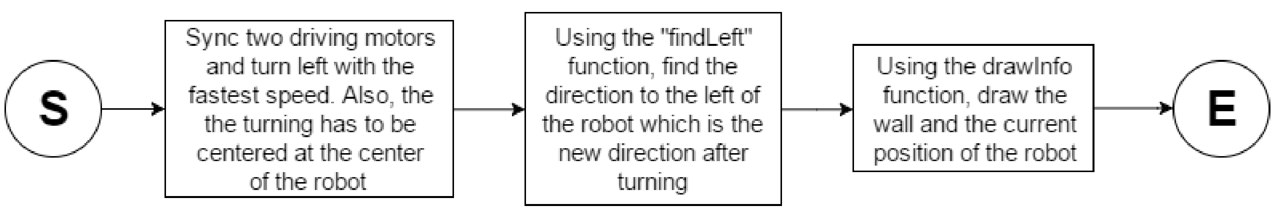

Figure 16: Flow Chart for Turning Left

  - 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();
```

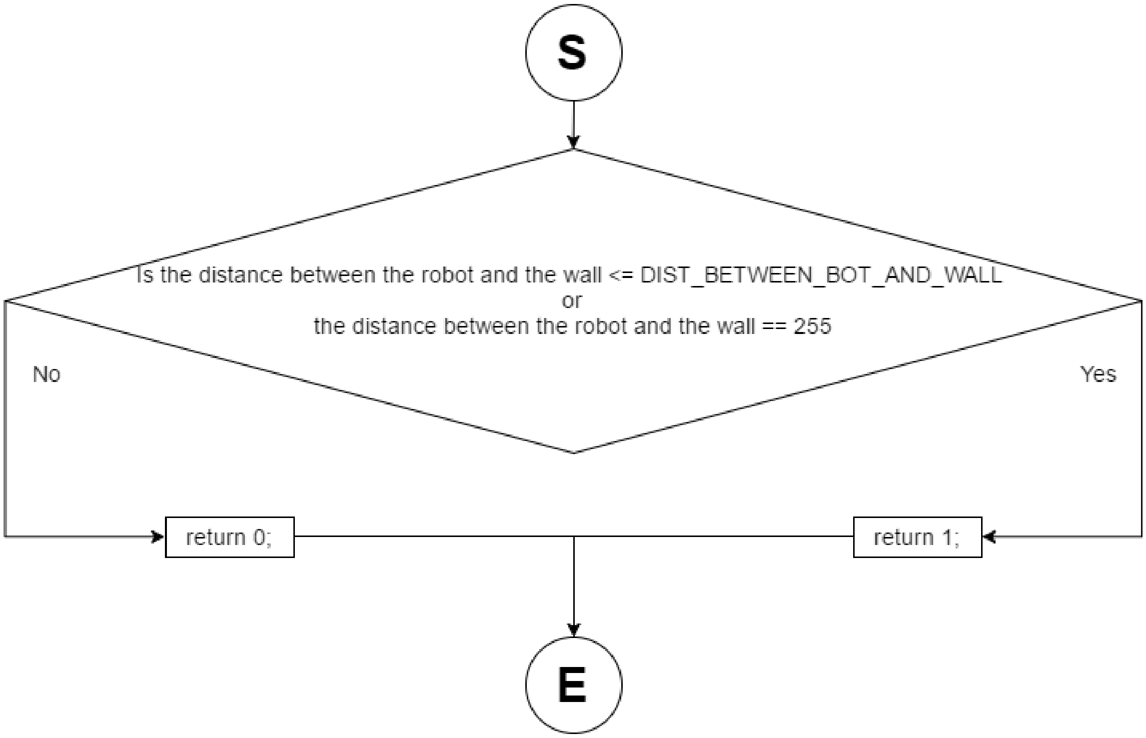

Figure 17: Flow Chart for Wall Detecting Function

- 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);`

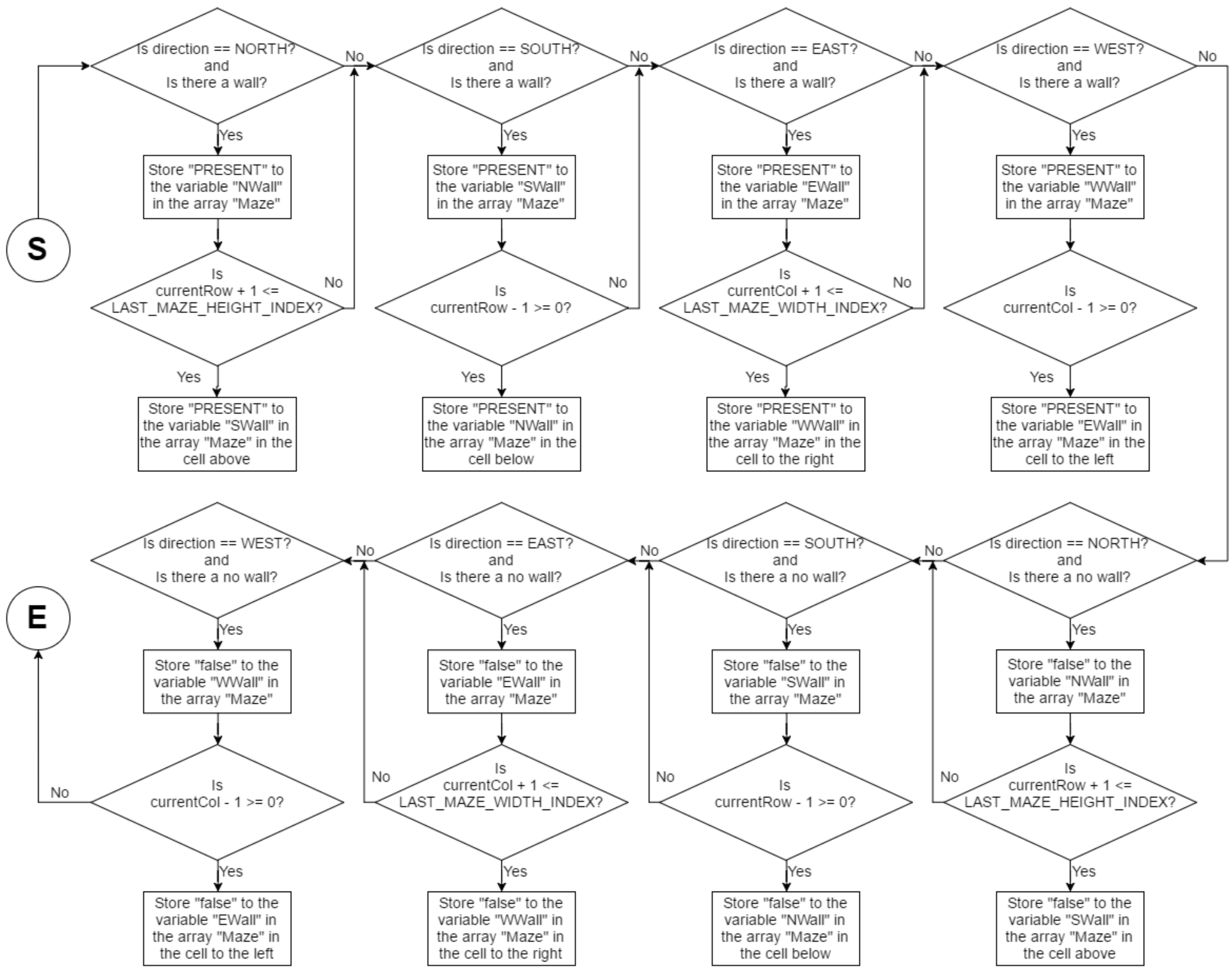

Figure 18: Flow Chart for Storing Data Function

  - 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);
```

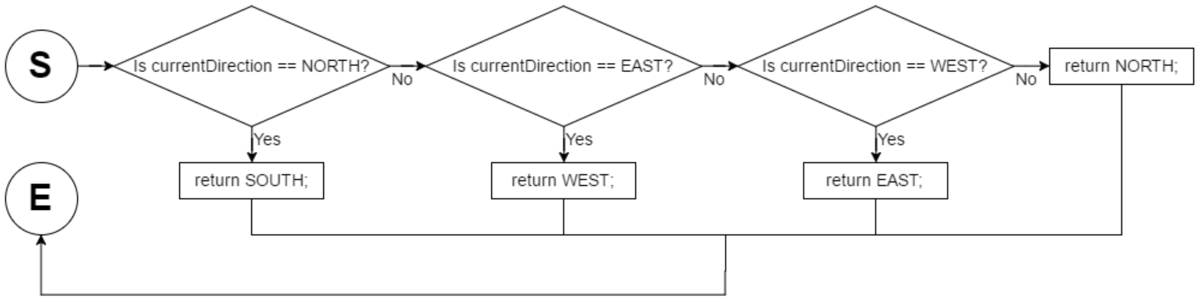

Figure 19: Flow Chart for Finding Back Function

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

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

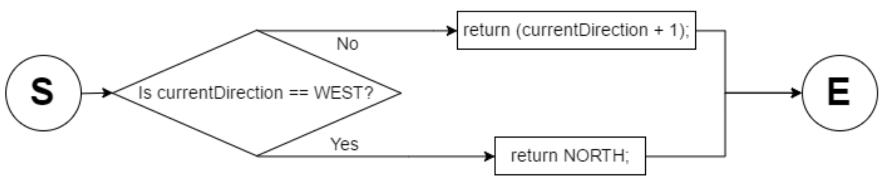

Figure 20: Flow Chart for Finding Right Function

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

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

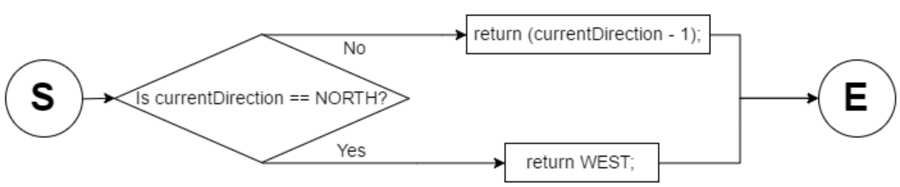

Figure 21: Flow Chart for Finding Left Function

  - 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);
```

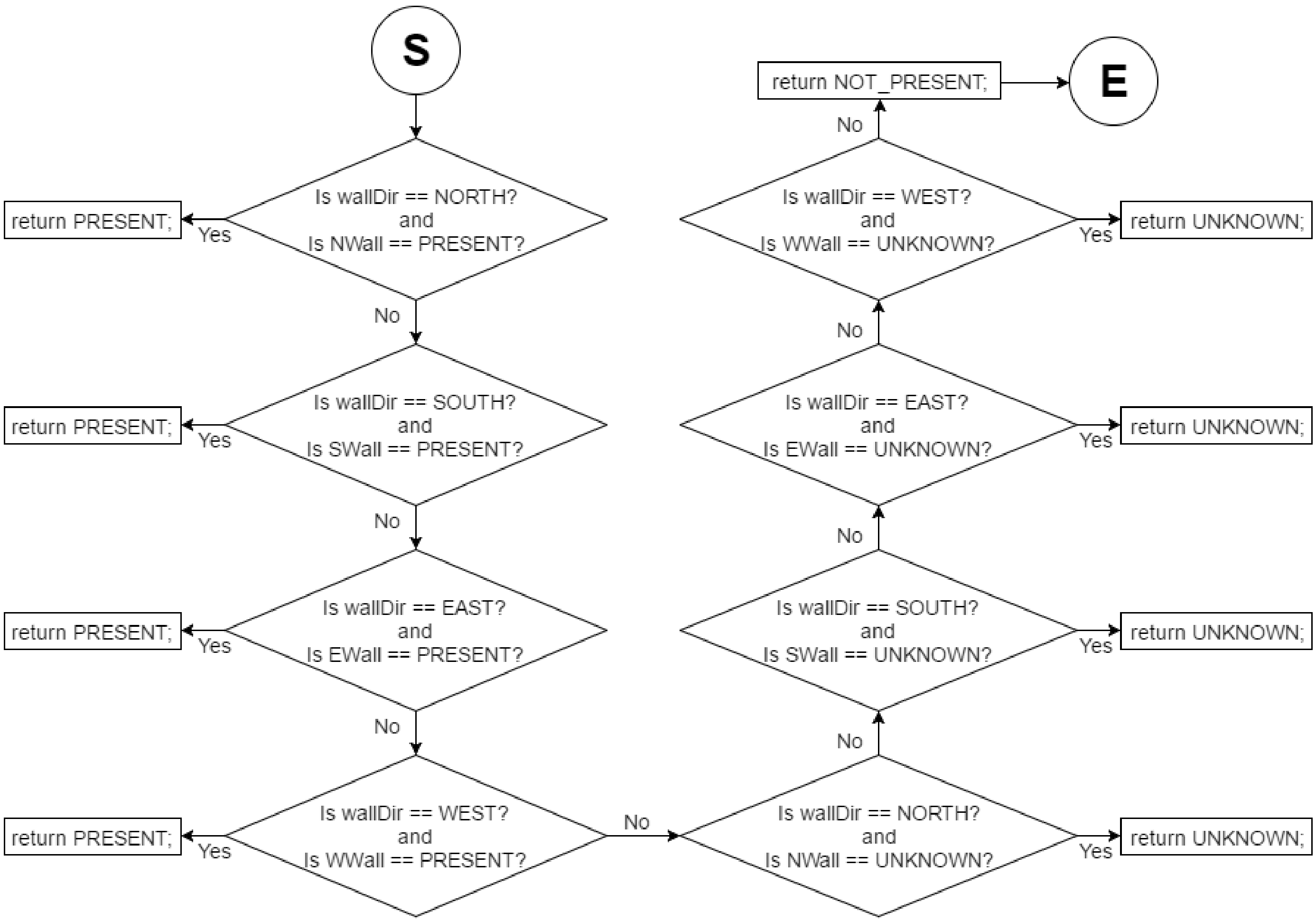

Figure 22: Flow Chart for Finding Existence of Wall from the data

- 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);
  ```

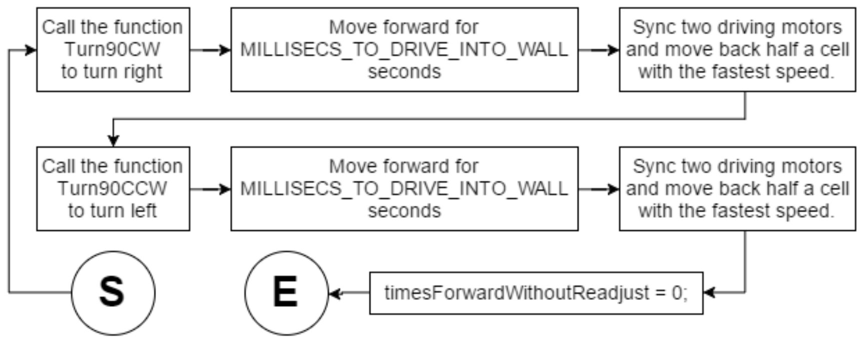

Figure 23: Flow Chart for Readjusting using Front wall and Right wall

- 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);
  ```

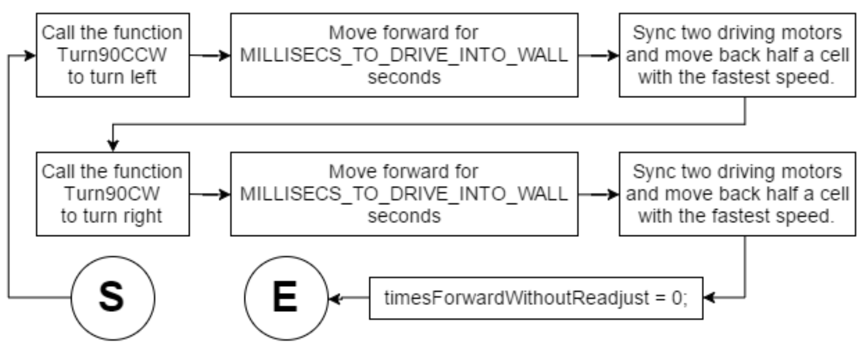

Figure 24: Flow Chart for Readjusting using Back wall and Left wall

  - 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);
```

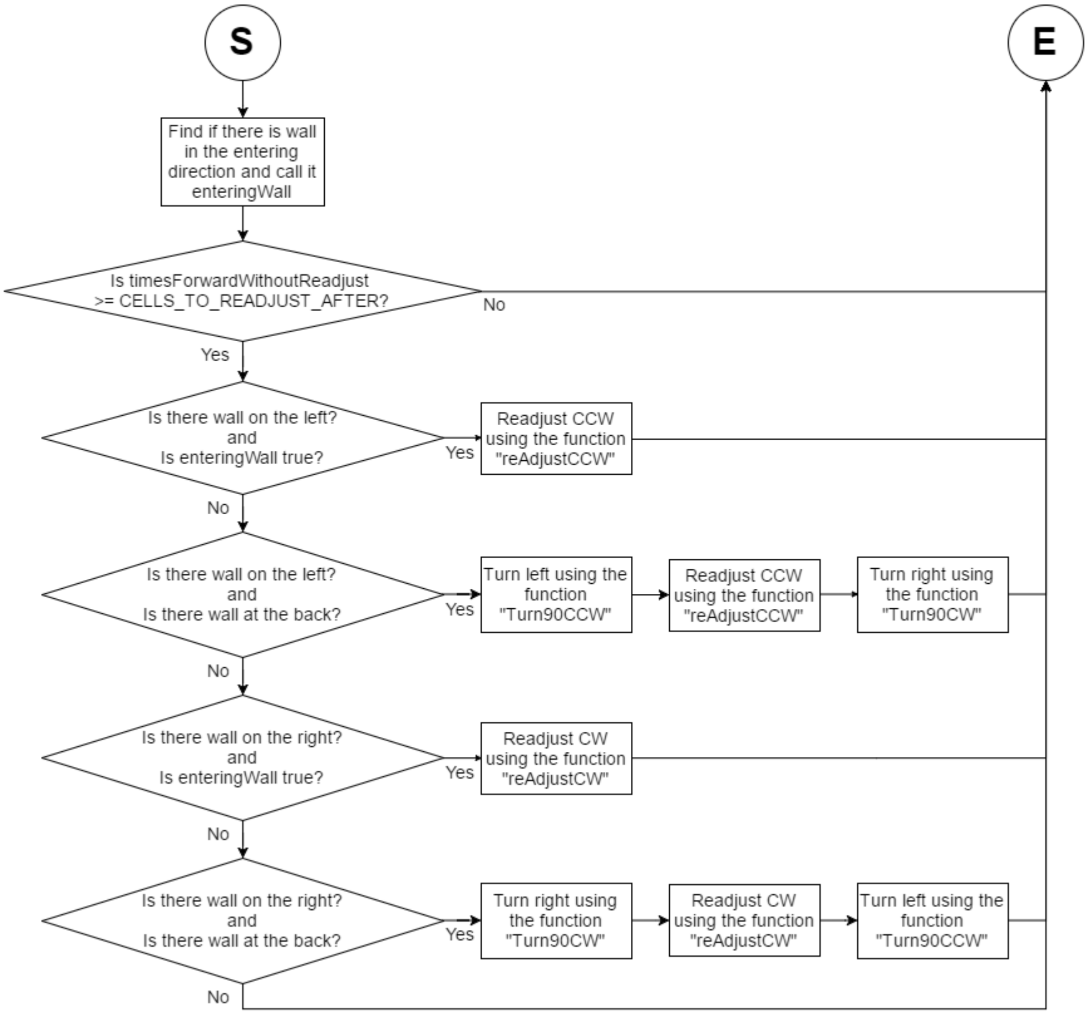

Figure 25: Flow Chart for Readjusting using walls detected

- 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);
  ```

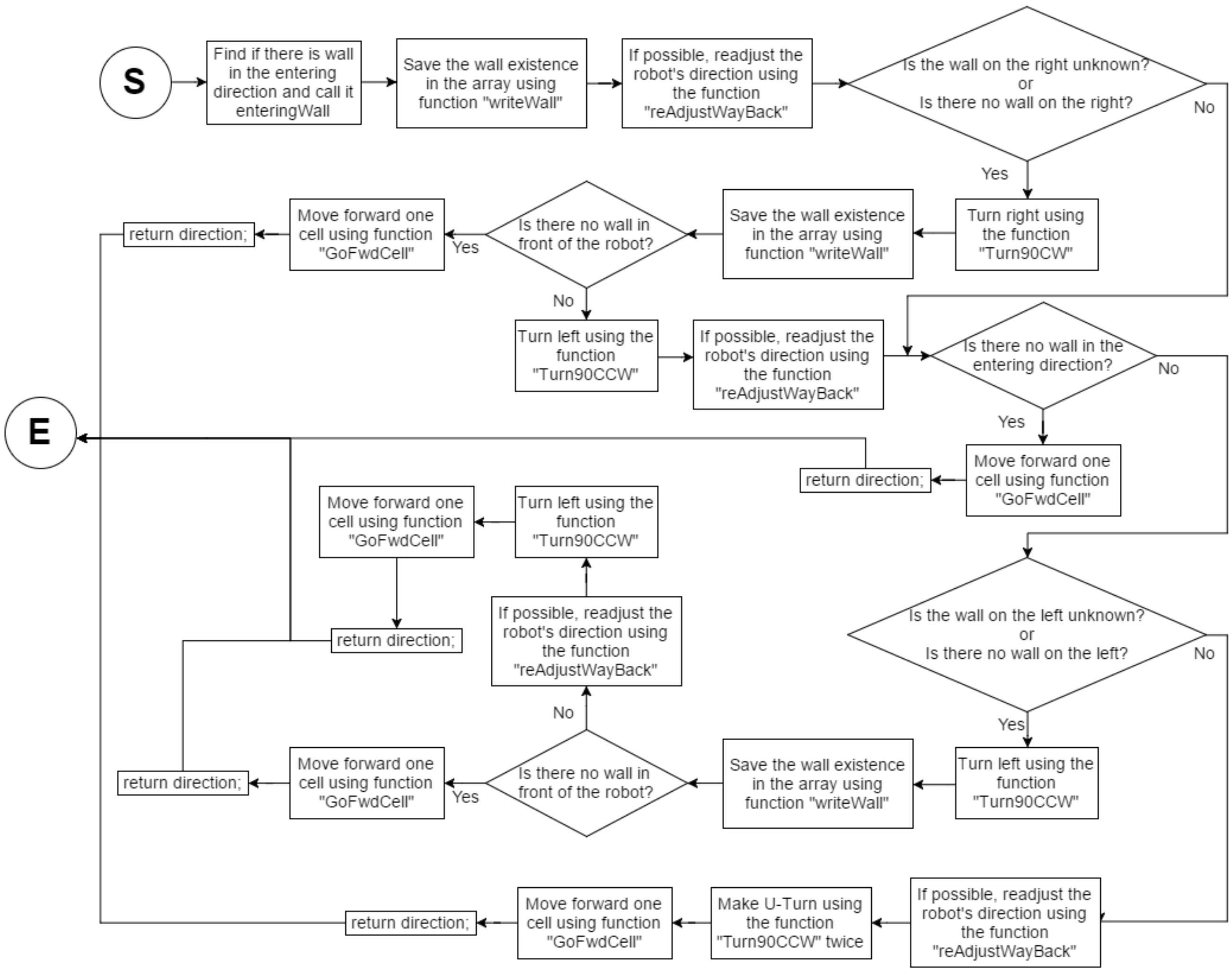

Figure 26: Flow Chart for Movement Function

  - 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();
```

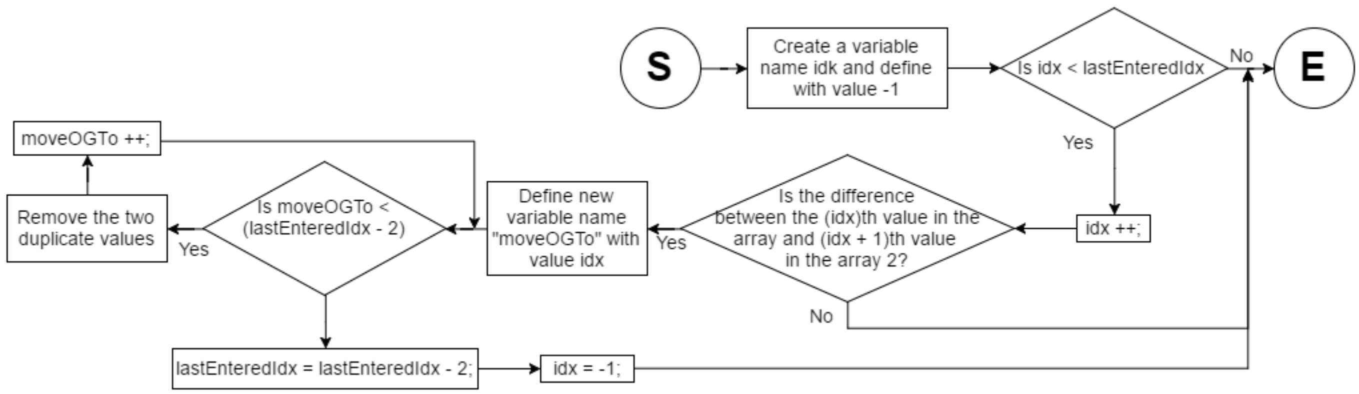

Figure 27: Flow Chart for Deleting Duplicate Function

  - 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();
```

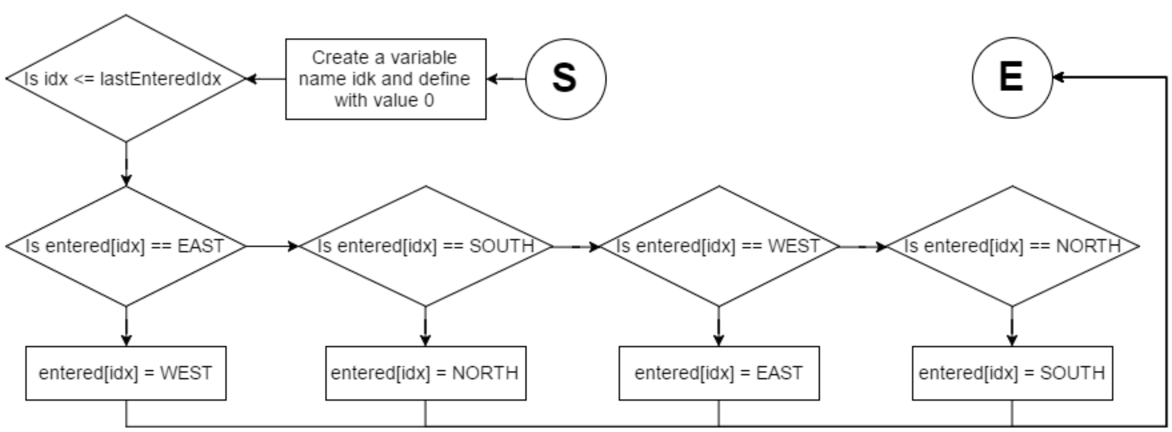

Figure 28: Flow Chart for Reversing Order of Values in the Array

  - 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);
```

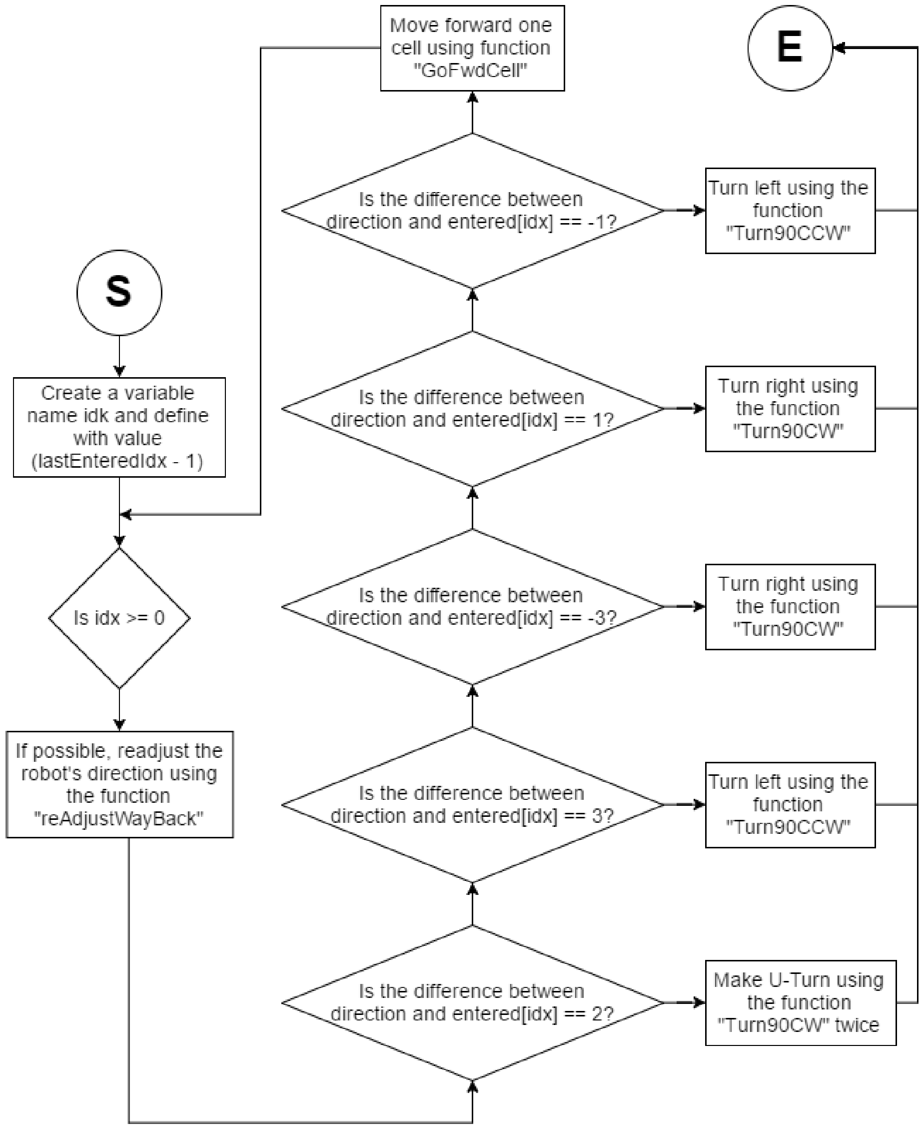

Figure 29: Flow Chart for Function to Go Back to Initial Position

- 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:

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

- Flow chart is on the next page.

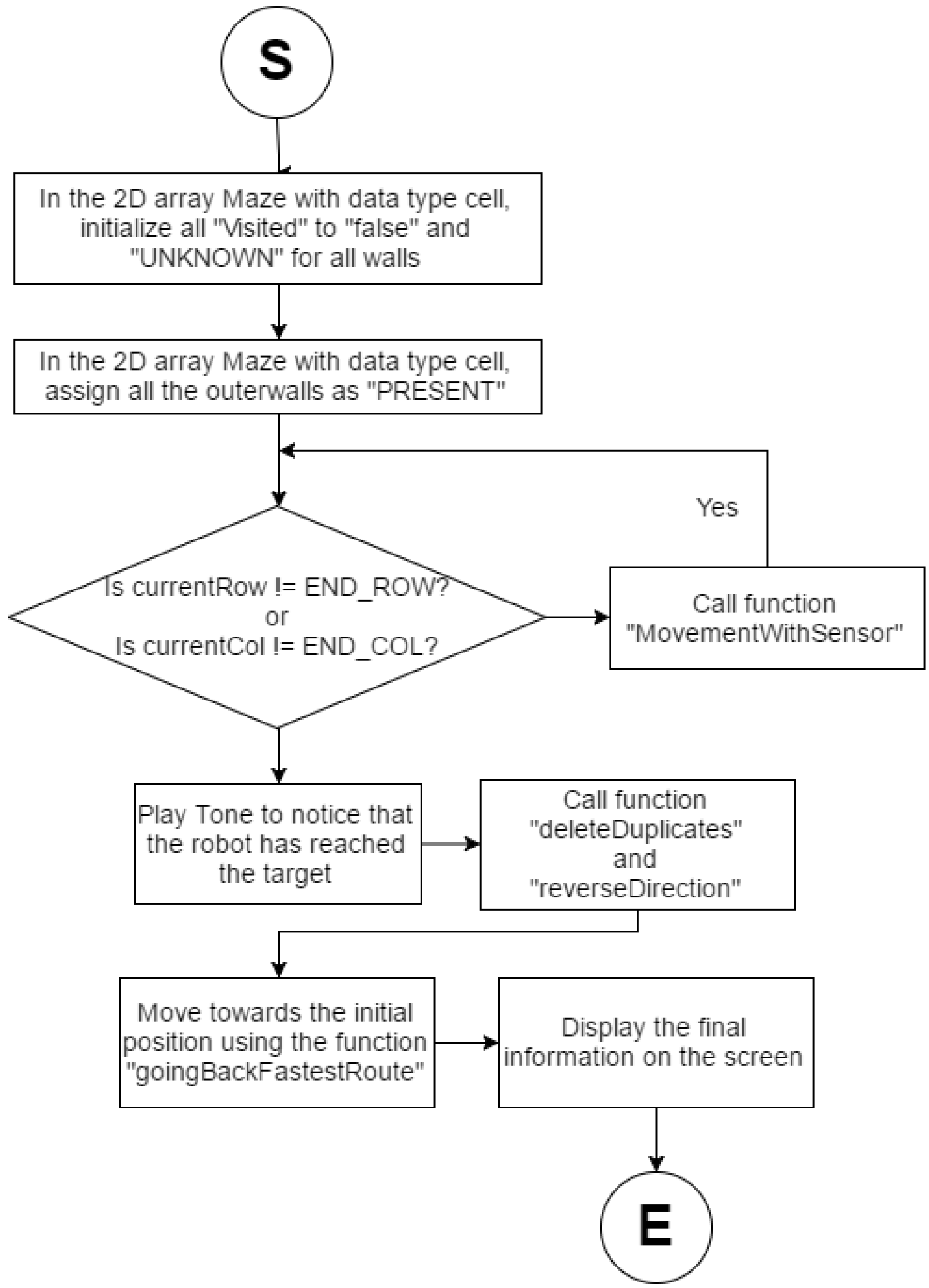

Figure 30: Flow Chart of the Main Function

4 Appendix

4.1 Full Source Code

```
#pragma config(Sensor, S1, distance, sensorEV3_Ultrasonic)
#pragma config(Motor, motorA, leftDrive, tmotorEV3_Large, PIDControl, driveLeft, encoder)
#pragma config(Motor, motorD, rightDrive, tmotorEV3_Large, PIDControl, driveRight, encoder)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

// Constants for robot's knowledge
#define NOT_PRESENT 0
#define PRESENT 1
#define UNKNOWN 2

// Maximum distance between robot and the wall
float const DIST_BETWEEN_BOT_AND_WALL = 7.6;

// Define directions using numbers
#define NORTH 0
#define EAST 1
#define SOUTH 2
#define WEST 3

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

// Starting and End positions defined with Row and Column numbers
// These positions were used for the Demo
int const START_ROW = 2;
int const START_COL = 0;
int const END_ROW = 3;
int const END_COL = 4;
// (3,0) to (3,4) - longest route was the longest path for the practice

// Current position defined
int currentRow = START_ROW;
int currentCol = START_COL;

// Constants for beeping mechanism
int const MILI_TO_BEEP_FOR = 200;
int const FREQUENCY = 300;

// Uncertainty due to property of integer division of computer
float const UNCERTAINTY_STRAIGHT = 19;
float const UNCERTAINTY_ROT = 27;
```

```
float const UNCERTAINTY_READJUST = 35;

// Movement Variabels defined
float const ONE_ROTATION = 360 + UNCERTAINTY_STRAIGHT;
float const QUARTER_ROTATION = 180 + UNCERTAINTY_ROT;
float const SIZE_OF_ONE_CELL = 22.5425; //cm
float const DRIVE_GEAR_RATIO = 5;
float const DIAMETER_OF_WHEEL = 5.5; // cm
float const CIRCUMFERENCE_OF_WHEEL = PI * DIAMETER_OF_WHEEL;

// Speed Variable
int const FORWARD = -100;
int const BACKWARD = -FORWARD;

// MAZE VARIABLES
int const MAZE_WIDTH = 6;
int const MAZE_HEIGHT = 4;
int const LAST_MAZE_HEIGHT_INDEX = MAZE_HEIGHT - 1;
int const LAST_MAZE_WIDTH_INDEX = MAZE_WIDTH - 1;
cell Maze[MAZE_HEIGHT][MAZE_WIDTH];

// Array to save up how the robot entered each cells
int entered[MAZE_WIDTH*MAZE_HEIGHT*4];
int lastEnteredIdx = 0;

// Constants for displaying mechanism
#define SCREEN_HEIGHT 127
#define SCREEN_WIDTH 177
#define CELL_HEIGHT (SCREEN_HEIGHT / MAZE_HEIGHT)
#define CELL_WIDTH (SCREEN_HEIGHT / MAZE_WIDTH)
#define CELL_HEIGHT_MIDDLE (CELL_HEIGHT / 2)
#define CELL_WIDTH_MIDDLE (CELL_WIDTH / 2)

// Constants for readjusting mechanism
int const MILLISECS_TO_DRIVE_INTO_WALL = 1100;
int const CELLS_TO_READJUST_AFTER = 3;
int timesForwardWithoutReadjust = 0;

// Call functions
void goFwdCell(int direction);
int Turn90CW(int direction);
int Turn90CCW(int direction);
int MovementWithSensor(int direction);
void reverseDirection();
void deleteDuplicates();
int goingBackFastestRoute(int direction);
void drawInfo(int direction);
void reAdjustCCW(int direction);
void reAdjustCW(int direction);
int findLeft(int currentDirection);
int findRight(int currentDirection);
```

```
int findBackDir (int currentDirection);
int isThereWallInDir(int wallDir);
void reAdjustWayBack(int direction);


void drawInfo(int direction){
    eraseDisplay();

    for(int r = 0; r < MAZE_HEIGHT; r++){
        for(int c = 0; c < MAZE_WIDTH; c++){

            if(Maze[r][c].SWall == PRESENT){
                drawLine(c*CELL_WIDTH,r*CELL_HEIGHT,c*CELL_WIDTH + CELL_WIDTH,r*CELL_HEIGHT);
            }
            if(Maze[r][c].NWall == PRESENT){
                drawLine(c*CELL_WIDTH,r*CELL_HEIGHT + CELL_HEIGHT,
                         c*CELL_WIDTH + CELL_WIDTH,r*CELL_HEIGHT + CELL_HEIGHT);
            }
            if(Maze[r][c].WWall == PRESENT){
                drawLine(c*CELL_WIDTH,r*CELL_HEIGHT,c*CELL_WIDTH,
                         r*CELL_HEIGHT + CELL_HEIGHT);
            }
            if(Maze[r][c].EWall == PRESENT){
                drawLine(c*CELL_WIDTH + CELL_WIDTH,r*CELL_HEIGHT,
                         c*CELL_WIDTH + CELL_WIDTH, r*CELL_HEIGHT + CELL_HEIGHT);
            }
        }
    }

    if(direction == NORTH){
        displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
                           currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "^");
    }
    else if(direction == EAST){
        displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
                           currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, ">");
    }
    else if(direction == WEST){
        displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
                           currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "<");
    }
    else if(direction == SOUTH){
        displayBigStringAt(currentCol*CELL_WIDTH + CELL_WIDTH_MIDDLE,
                           currentRow*CELL_HEIGHT + CELL_HEIGHT_MIDDLE, "v");
    }
}
```

```
task main(){

    for (int c = 0; c < MAZE_WIDTH; c++){
        for (int r = 0; r < MAZE_HEIGHT; r++){
            Maze[r][c].Visited = false;
            Maze[r][c].NWall = UNKNOWN;
            Maze[r][c].SWall = UNKNOWN;
            Maze[r][c].EWall = UNKNOWN;
            Maze[r][c].WWall = UNKNOWN;
        }
    }

    // Assigning walls [row][col]
    for (int c = 0; c < MAZE_WIDTH; c++){
        Maze[0][c].SWall = PRESENT;
        Maze[LAST_MAZE_HEIGHT_INDEX][c].NWall = PRESENT;
    }

    for (int r = 0; r < MAZE_HEIGHT; r++){
        Maze[r][0].WWall = PRESENT;
        Maze[r][LAST_MAZE_WIDTH_INDEX].EWall = PRESENT;
    }

    int direction = NORTH;

    Maze[currentRow][currentCol].entryDir = direction;
    Maze[currentRow][currentCol].Visited = true;

    while(currentRow != END_ROW || currentCol != END_COL){
        direction = MovementWithSensor(direction);
        entered[lastEnteredIdx] = direction;
        lastEnteredIdx++;
    }

    playTone(FREQUENCY, MILI_TO_BEEP_FOR);

    deleteDuplicates();

    sleep(MILI_TO_BEEP_FOR * 10);

    reverseDirection();

    direction = goingBackFastestRoute(direction);

    drawInfo(direction);

    sleep(390000);
}
```

```
    void deleteDuplicates(){
        int idx = -1;

        while(idx < lastEnteredIdx){
            idx++;

            if(abs(entered[idx] - entered[idx + 1]) == 2){
                for(int moveOGTo = idx; moveOGTo <= lastEnteredIdx - 2; moveOGTo++){
                    entered[moveOGTo] = entered[moveOGTo + 2];
                }
                lastEnteredIdx = lastEnteredIdx - 2;
                idx = -1;
            }
        }
    }


    void reverseDirection(){
        for(int idx = 0; idx <= lastEnteredIdx; idx++){
            if(entered[idx]==EAST){
                entered[idx] = WEST;
            }
            else if(entered[idx]==SOUTH){
                entered[idx] = NORTH;
            }
            else if(entered[idx]==WEST){
                entered[idx] = EAST;
            }
            else if(entered[idx]==NORTH){
                entered[idx] = SOUTH;
            }
        }
    }


    int goingBackFastestRoute(int direction){

        for(int idx = lastEnteredIdx - 1; idx >= 0; idx--){
            reAdjustWayBack(direction);
            int turnNum = entered[idx] - direction;

            if(abs(turnNum) == 2){
                direction = Turn90CW(direction);
                direction = Turn90CW(direction);
            }
            else if(turnNum == 3){
                direction = Turn90CCW(direction);
            }
            else if(turnNum == -3){
                direction = Turn90CW(direction);
            }
```

```
            else if(turnNum == 1){
                direction = Turn90CW(direction);
            }
            else if(turnNum == -1){
                direction = Turn90CCW(direction);
            }
            goFwdCell(direction);
        }
        return direction;
    }


    void goFwdCell(int direction){
        setMotorSyncEncoder(leftDrive, rightDrive, 0,
                            (SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
                            * ONE_ROTATION, FORWARD);

        repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){

        }
        if (direction == NORTH){
            Maze[currentRow][currentCol].NWall = false;
            currentRow++;
            Maze[currentRow][currentCol].SWall = false;
        }
        else if (direction == SOUTH){
            Maze[currentRow][currentCol].SWall = false;
            currentRow--;
            Maze[currentRow][currentCol].NWall = false;
        }
        else if (direction == EAST){
            Maze[currentRow][currentCol].EWall = false;
            currentCol++;
            Maze[currentRow][currentCol].WWall = false;
        }
        else if (direction == WEST){
            Maze[currentRow][currentCol].WWall = false;
            currentCol--;
            Maze[currentRow][currentCol].EWall = false;
        }

        Maze[currentRow][currentCol].entryDir = direction;
        Maze[currentRow][currentCol].Visited = true;

        timesForwardWithoutReadjust++;
    }
```

```
int Turn90CCW(int direction){
    setMotorSyncEncoder(leftDrive, rightDrive, -100, QUARTER_ROTATION * DRIVE_GEAR_RATIO,
                        FORWARD);

    repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){

    }
    direction = findLeft(direction);
    drawInfo(direction);
    return direction;
}


int Turn90CW(int direction){
    setMotorSyncEncoder(leftDrive, rightDrive, 100, QUARTER_ROTATION * DRIVE_GEAR_RATIO,
                        FORWARD);

    repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){

    }
    direction = findRight(direction);
    drawInfo(direction);
    return direction;
}


int thereIsWall(){
    if(getUSDistance(distance)<=DIST_BETWEEN_BOT_AND_WALL || getUSDistance(distance)==255){
        return 1;
    }
    return 0;
}


void writeWall(int direction){
    if(direction == NORTH && thereIsWall()){
        Maze[currentRow][currentCol].NWall = PRESENT;
        if(currentRow + 1 <= LAST_MAZE_HEIGHT_INDEX){
            Maze[currentRow + 1][currentCol].SWall = PRESENT;
        }
    }
    else if(direction == SOUTH && thereIsWall()){
        Maze[currentRow][currentCol].SWall = PRESENT;
        if(currentRow - 1 >= 0){
            Maze[currentRow - 1][currentCol].NWall = PRESENT;
        }
    }
    else if(direction == EAST && thereIsWall()){
        Maze[currentRow][currentCol].EWall = PRESENT;
        if(currentCol + 1 <= LAST_MAZE_WIDTH_INDEX){
            Maze[currentRow][currentCol + 1].WWall = PRESENT;
```

```
            }
        }
        else if(direction == WEST && thereIsWall()){
            Maze[currentRow][currentCol].WWall = PRESENT;
            if(currentCol - 1 >= 0){
                Maze[currentRow][currentCol - 1].EWall = PRESENT;
            }
        }
        else if(direction == NORTH && !thereIsWall()){
            Maze[currentRow][currentCol].NWall = false;
            if(currentRow + 1 <= LAST_MAZE_HEIGHT_INDEX){
                Maze[currentRow + 1][currentCol].SWall = false;
            }
        }
        else if(direction == SOUTH && !thereIsWall()){
            Maze[currentRow][currentCol].SWall = false;
            if(currentRow - 1 >= 0){
                Maze[currentRow - 1][currentCol].NWall = false;
            }
        }
        else if(direction == EAST && !thereIsWall()){
            Maze[currentRow][currentCol].EWall = false;
            if(currentCol + 1 <= LAST_MAZE_WIDTH_INDEX){
                Maze[currentRow][currentCol + 1].WWall = false;
            }
        }
        else if(direction == WEST && !thereIsWall()){
            Maze[currentRow][currentCol].WWall = false;
            if(currentCol - 1 >= 0){
                Maze[currentRow][currentCol - 1].EWall = false;
            }
        }
    }


    // Checking order, North(0), East(1), West(3) then South(2)
    // right, north, left, back
    int MovementWithSensor(int direction){

        int enteringDirectionWall = thereIsWall();
        writeWall(direction);
        reAdjustWayBack(direction);

        // turn to check if wall is right
        if(isThereWallInDir(findRight(direction)) == UNKNOWN
           || isThereWallInDir(findRight(direction)) == NOT_PRESENT){
            direction = Turn90CW(direction);
            writeWall(direction);

            // go right if no wall right
            if(!thereIsWall()){
```

```
                goFwdCell(direction);
                return direction;
            }
            else{
                direction = Turn90CCW(direction);
                reAdjustWayBack(direction);
            }
        }
        if(!enteringDirectionWall){
                goFwdCell(direction);
                return direction;
        }

        // At this point, we know there r walls on the R and N
        // We are facing N
        // if we know there is wall left, go thru back
        if(isThereWallInDir(findLeft(direction)) == UNKNOWN
           || isThereWallInDir(findLeft(direction)) == NOT_PRESENT){

            direction = Turn90CCW(direction);
            writeWall(direction);

            if(!thereIsWall()){
                goFwdCell(direction);
                return direction;
            }
            else{
                reAdjustWayBack(direction);
                direction = Turn90CCW(direction);
                goFwdCell(direction);
                return direction;
            }
        }
        else{
            reAdjustWayBack(direction);
            direction = Turn90CCW(direction);
            direction = Turn90CCW(direction);
            goFwdCell(direction);
            return direction;
        }
        sleep(1000000);
    }
```

```
void reAdjustCCW(int direction){

    direction = Turn90CCW(direction);
    motor[rightDrive] = FORWARD;
    motor[leftDrive] = FORWARD;
    sleep(MILLISECS_TO_DRIVE_INTO_WALL);

    setMotorSyncEncoder(leftDrive, rightDrive, 0, ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)
                        *DRIVE_GEAR_RATIO * ONE_ROTATION)/7 + UNCERTAINTY_READJUST,
                         BACKWARD);

    repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){

    }
    Turn90CW(direction);
    motor[rightDrive] = FORWARD;
    motor[leftDrive] = FORWARD;
    sleep(MILLISECS_TO_DRIVE_INTO_WALL);

    setMotorSyncEncoder(leftDrive, rightDrive, 0,
                        ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
                        * ONE_ROTATION)/7 + UNCERTAINTY_READJUST, BACKWARD);

    repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){

    }
    timesForwardWithoutReadjust = 0;
}


void reAdjustCW(int direction){

    direction = Turn90CW(direction);
    motor[rightDrive] = FORWARD;
    motor[leftDrive] = FORWARD;
    sleep(MILLISECS_TO_DRIVE_INTO_WALL);

    setMotorSyncEncoder(leftDrive, rightDrive, 0,
                        ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
                        * ONE_ROTATION)/7 + UNCERTAINTY_READJUST, BACKWARD);

    repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){

    }
    Turn90CCW(direction);
    motor[rightDrive] = FORWARD;
    motor[leftDrive] = FORWARD;
    sleep(MILLISECS_TO_DRIVE_INTO_WALL);

    setMotorSyncEncoder(leftDrive, rightDrive, 0,
                        ((SIZE_OF_ONE_CELL / CIRCUMFERENCE_OF_WHEEL)*DRIVE_GEAR_RATIO
```

```
                              * ONE_ROTATION)/7 + UNCERTAINTY_READJUST, BACKWARD);

    repeatUntil(!getMotorRunning(leftDrive) && !getMotorRunning(rightDrive)){

    }
    timesForwardWithoutReadjust = 0;
}


void reAdjustWayBack(int direction){
    int enteringWall = thereIsWall();

    if(timesForwardWithoutReadjust >=  CELLS_TO_READJUST_AFTER){
        if(isThereWallInDir(findLeft(direction)) == PRESENT && enteringWall){
            reAdjustCCW(direction);
        }
        else if(isThereWallInDir(findLeft(direction)) == PRESENT &&
                isThereWallInDir(findBackDir(direction)) == PRESENT){
            direction = Turn90CCW(direction);
            reAdjustCCW(direction);
            direction = Turn90CW(direction);
        }
        else if(enteringWall && isThereWallInDir(findRight(direction)) == PRESENT){
            reAdjustCW(direction);
        }
        else if(isThereWallInDir(findRight(direction)) == PRESENT &&
                isThereWallInDir(findBackDir(direction)) == PRESENT){
            direction = Turn90CW(direction);
            reAdjustCW(direction);
            direction = Turn90CCW(direction);
        }
    }
}


int findBackDir (int currentDirection){
    if(currentDirection == NORTH){
        return SOUTH;
    }
    else if(currentDirection == EAST){
        return WEST;
    }
    else if(currentDirection == WEST){
        return EAST;
    }
    return NORTH;
}
```

```
int findRight(int currentDirection){

    if(currentDirection == WEST){
        return NORTH;
    }
    else{
        return currentDirection + 1;
    }
}


int findLeft(int currentDirection){

    if(currentDirection == NORTH){
        return WEST;
    }
    else{
        return currentDirection - 1;
    }
}


int isThereWallInDir(int wallDir){
    if(wallDir == NORTH && Maze[currentRow][currentCol].NWall == PRESENT){
        return PRESENT;
    }
    else if(wallDir == SOUTH && Maze[currentRow][currentCol].SWall == PRESENT){
        return PRESENT;
    }
    else if(wallDir == EAST && Maze[currentRow][currentCol].EWall == PRESENT){
        return PRESENT;
    }
    else if(wallDir == WEST && Maze[currentRow][currentCol].WWall == PRESENT){
        return PRESENT;
    }

    if(wallDir == NORTH && Maze[currentRow][currentCol].NWall == UNKNOWN){
        return UNKNOWN;
    }
    else if(wallDir == SOUTH && Maze[currentRow][currentCol].SWall == UNKNOWN){
        return UNKNOWN;
    }
    else if(wallDir == EAST && Maze[currentRow][currentCol].EWall == UNKNOWN){
        return UNKNOWN;
    }
    else if(wallDir == WEST && Maze[currentRow][currentCol].WWall == UNKNOWN){
        return UNKNOWN;
    }
    return NOT_PRESENT;
}
```

4.2 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.