SFU

MSE110

Final Report

Submission Date:
/ / 2016

Group | | |

members

First name	Last name	Student Number

Table of Contents

Table of Figures

Introduction

The purpose of this project is to explore the maze solving capability of the robot in the given maze. The robot will be starting at any given cell and scanning the maze to a target cell. After reaching the target cell, the robot returns to the starting cell within the shortest path. The developed algorithm enables the robot to save the state of the cells in a 2D array structure. This data structure includes the orientation of the current robot, the blocked or unblocked side of the wall, and visited or unvisited cell. The robot uses the ultrasound sensor to detect the wall in front and save the state of each cell it passes through. The robot optimizes the minimum pathway to the initial position after reaching the target cell. Having a basic understanding of the 2D array data structure, as well as this robot C coding algorithm, is necessary in order to be able to solve for these functions.
This report describes the mechanical structure, as well as software design we used to solve the maze and discussion on the problem and the corresponding solution. The problem involves the alignment issue that the disposition from the center of the cell as the number of the passing cell increases. The purpose of this report is to illustrate the detailed design group 23 has carried throughout the project.

Progress Report

In this section, we listed the weekly progress on the development on the maze solving robot.

Week 1

We determined the hardware constraints and the necessary equipment for the maze-solving robot. We constructed the main part of the robot by connecting two large motors with the brick. Also, we added Lego pieces to stabilize and strengthen the robot. Afterward, we observed the size of robot whether it is suitable for the maze.

Week 2

We finished up the robot's mechanical structure by adding ultrasound and touch sensors. We wrote basic functions on RobotC such as turn right to test if the robot can perform tasks properly with the given mechanical structure. Then, we created several functions to perform the basic movements such as turn right, turn left, go forward and stop motor. In addition, we wrote the first section of maze-solving algorithm, which is the robot moving from a starting cell to a target cell using the right wall following algorithm.

Week 3

We completed the maze-solving program by writing the code for the robot to trace back to the starting cell within the shortest path. Moreover, we wrote an extra code to align the robot to the center of the cell.

Week 4

We wrote the last part of maze-solving program which allow user to enter the row and column of the starting and the target cell. To make the code more robust, we tested our maze-solving robot with different starting cell and target cell, and modified the code as we encountered problems.

Mechanical Design

Size of robot

Since the robot has to traverse through the maze, the size of robot will have a significant effect on the robot performance. Therefore, the robot has to be smaller than the size of one cell in the maze $8\frac{7}{8}'' \times 8\frac{7}{8}''$. We decided to place the brick vertically to reduce the width of the robot's body, and we connected 2 large motors on the back of the brick for medium size wheels.

Axis of rotation

Due to the limited space of one cell, the robot's axis of rotation should be in the center. Therefore, the wheel should be placed in the center of robot body. Moreover, the distance between two wheels should not be larger than the body's width to remain the small size robot.

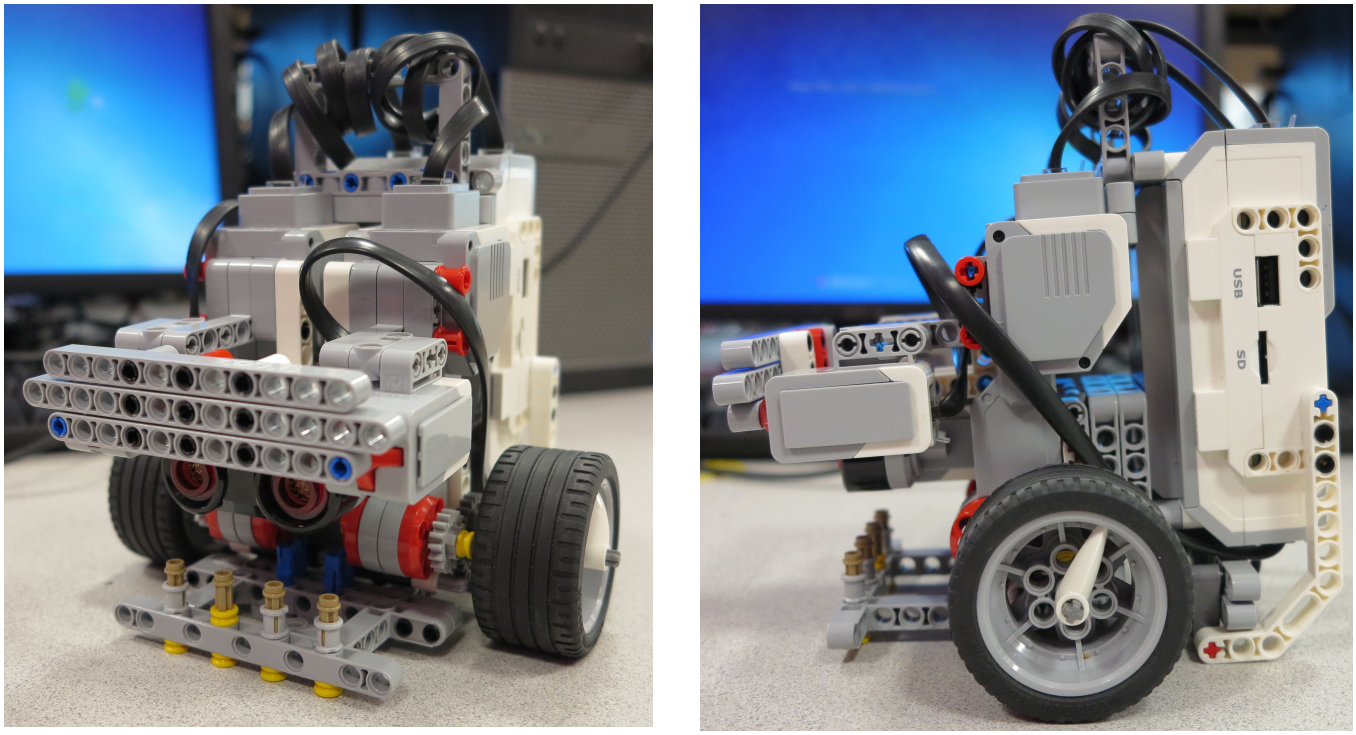

Figure 1: Maze solving robot the overall and the side view

Gear ratio

The maze-solving robot turns in various directions: left-turn, right-turn and U-turn. In order to obtain accurate 90 degrees and 180 degrees turns, we use gear ratio of 24:16. We connected 16-tooth gears to the motor in both side and connected 24-tooth gears to the wheels. Since there are energy loss and backlash due to a friction, we use trial and error method to obtain the right encoder value of 90 degrees turn and moving forward.

Table 1: Motor encoder value and the corresponding response from the motor

	Trial	MotorA (left)	MotorD (right)	Result
Go Forward (horizontal)	1	700	700	The robot could not reach the center of the cell.
	2	725	725	
	3	750	750	The robot could go from a center of one cell to another. However, the robot could not move perfectly straight.
	4	745	750	The robot could move straight from a center of one cell to a center of another cell.
Go Forward (vertical)	1	560	560	The robot could not reach the center of the cell.
	2	670	670	
	3	705	710	The robot could move straight. However, it could not reach the center of another cell
	4	700	705	The robot could move straight from a center of one cell to a center of another cell
Turn Right	1	273	-273	The robot turned approximately 120 degrees
	2	90*3.15	-90*3.15	
	3	90*3.05	-90*3.05	The robot turned slightly exceed 90 degrees
	4	90*3.02	90*3.02	The robot could turn approximately 90 degrees
Turn Left	1	-273	273	The robot turned approximately 120 degrees
	2	-90*3.15	90*3.15	
	3	-90*2.85	90*2.85	The robot turned slightly less than 90 degrees
	4	-90*2.98	90*2.98	The robot could turn approximately 90 degrees

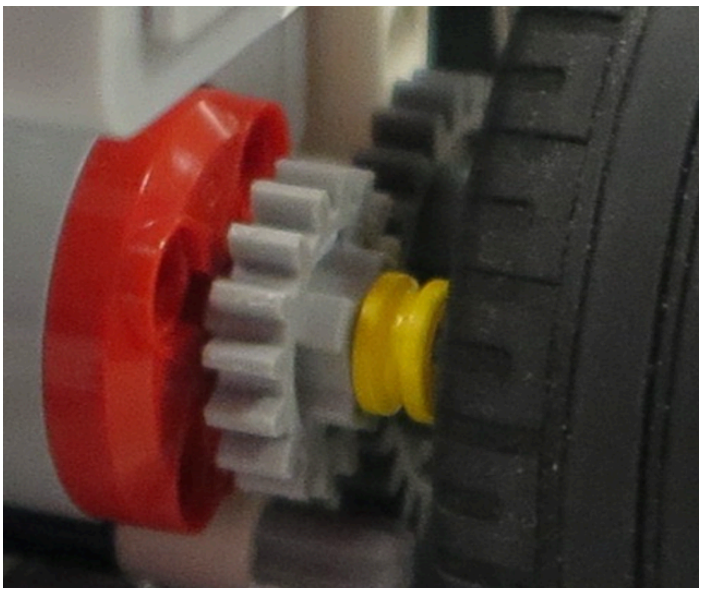

Figure 2: 24:16 Gear ratio

Wall detection

Ultrasonic is a device that returns the distance between the device itself and the obstacle in front of it. So, we placed an ultrasonic in the front of robot for detecting the wall. To acquire a suitable wall-detecting threshold, we tried different values of the distance from the robot to a wall, and observed the robot's performance.

Table 2: Threshold settings and results

Trail	Threshold (cm)	Result
1	5	The robot could not detect some walls even though the robot is already in the center of a cell.
2	7	The robot could not detect a wall when it was not at the center of a cell.
3	14	The robot thought that there was a wall on that cell although the wall was located on the next cell.
4	8	Similarly to trail 2, the robot could not recognize that there is a wall when it was not exactly at the center of a cell
5	10	The robot could detect all of the wall in that cell and it did not over-detect the wall on the other cell

Realignment

After several trails, we discovered that the robot could not obtain a perfect turn. To solve this problem, we connected two touch sensors with beams in the front of robot. Once the robot reached the corner, the robot would bump into both side of walls and back up in certain distance to stay in the center of the cell.

Table 3: The change in threshold and the corresponding results

Trial	Threshold (horizontal)	Threshold (vertical)	Result
1	4.9	5.3	After bumping to the wall, the robot backed up farther than the center of the cell
2	4.3	4.9	The robot still backed up farther than the center of the cell
3	4.5	5.1	
4	4.3	4.9	
5	4	4.5	The robot backed up to the center of the cell

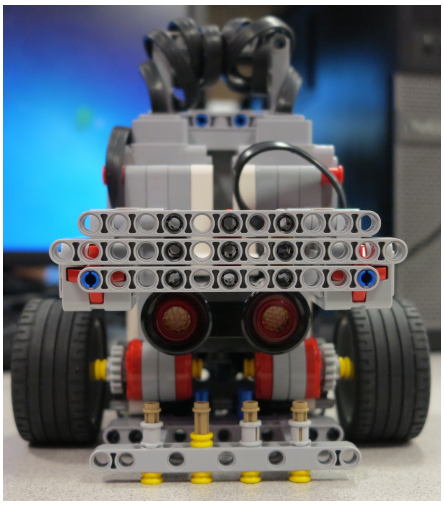

Figure 3: The front view of the maze-solving robot

Software Design

The maze-solving program contains two main section of code:

1. Solving the maze

In order to solve the maze, we used the right wall follower algorithm. This algorithm is for the robot to follow its right side of the wall until it reaches the target cell. We defined the orientation of the robot to start by facing the north direction. Once, the program starts, the robot would turn right and check if there is a wall. If there is no wall, the robot will move forward to the next cell. However, if there is a wall, the robot will keep turning left until it detects no wall and move forward to the next cell.

2. Moving back to home cell by using the shortest path

To evaluate the shortest path, we stored the moving directions of robot, which are north, south, east or west, into an array. Then, we eliminated the direction to the dead-end by checking for the opposite-direction pairs (north-south or east-west) and replaced that direction value with -1. For robot moving back to home, we inverse directions by changing north to south, east to west, south to north and west to east.

Table 4: Function descriptions

Function Name	Purpose
Cellinitial	To initialize the outer wall for the structure maze and marked every cell as an unvisited
ResetValue	To reset the encoder value for 2 large motors
TurnRight	For robot to turn right 90 degrees
TurnLeft	For robot to turn left 90 degrees
GoFwd	For robot to go forward one cell
Adjust	Help robot to realign by bumping into the wall

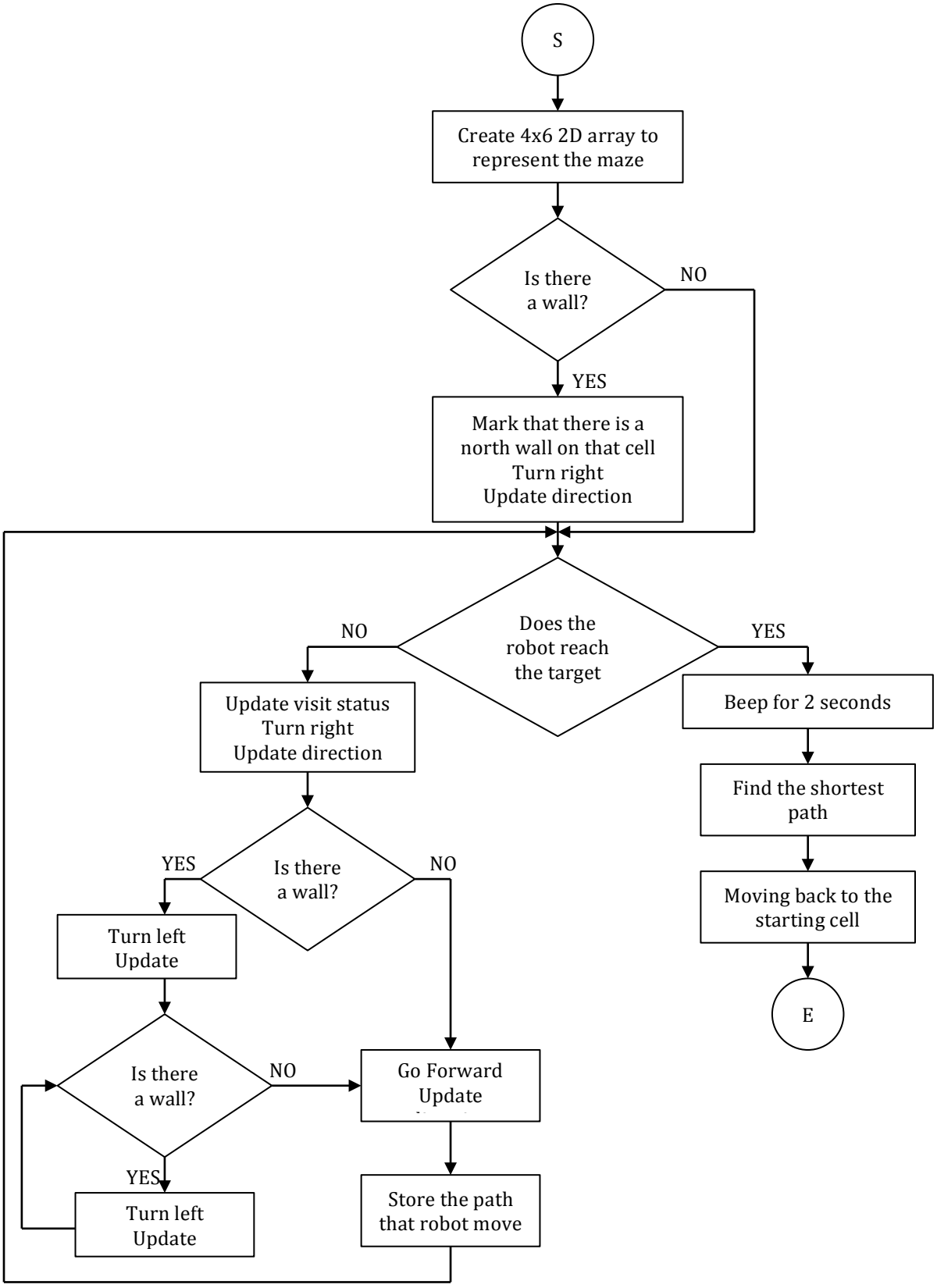

Figure 4: Flow chart diagram

Function Parameters

All of function does not take any argument, and they are void type. Therefore, they will not return any values.

Table 5: Global Variable

<table>
<tr><th>Variable Name</th><th>Type</th><th>Purpose</th></tr>
<tr><td>encodeValueMotorA</td><td>Integer</td><td>An encoder value of a left motor (motorA)</td></tr>
<tr><td>encodeValueMotorD</td><td>Integer</td><td>An encoder value of a right motor (motorD)</td></tr>
<tr><td>Walldistance</td><td>Integer</td><td>The maximum distance between the wall and the robot that the robot can detect (10 cm.)</td></tr>
<tr><td>direction</td><td>Integer</td><td>The direction that the robot faces:
0 – north, 1 – east, 2 – south, 3 – west</td></tr>
<tr><td>soundActive</td><td>Integer</td><td>A beep sound controller
- Defined as 0
- Assign 1 when the robot reaches the target cell</td></tr>
<tr><td>Maze</td><td>Array of structure</td><td>A group of data that contains information of each cell in the maze stores in 4x6 2D array:
<table>
<tr><th>Variable Name</th><th>Type</th><th>Purpose</th></tr>
<tr><td>NWall</td><td>Integer</td><td rowspan="4">To record whether there was a wall in each direction (North/East/South/West)</td></tr>
<tr><td>EWall</td><td>Integer</td></tr>
<tr><td>SWall</td><td>Integer</td></tr>
<tr><td>WWall</td><td>Integer</td></tr>
<tr><td>Visited</td><td>Character</td><td>To record whether the robot visited the cell</td></tr>
</table>
</td></tr>
</table>

Table 6: Local variables

Variable Name	Type	Purpose
ind	Integer	An index for an array (path) that store the direction that the robot moves from one cell to another
r_start	Integer	A position of row that the robot will start (home cell)
c_start	Integer	A position of column that the robot will start (home cell)
r_current	Integer	A current position of row that robot is on
c_current	Integer	A current position of column that robot is on
r_target	Integer	A position of row that the robot suppose to reach (cheese cell)
c_target	Integer	A position of column that the robot suppose to reach (cheese cell)
adjustActive	Integer	A controller for the robot to realign (related to Adjust function)
starting	Integer	An indicator that robot just start to move (related to Adjust function)
begin_index	Integer	A position of the first -1 that is located in the "path" array (related to shortest path evaluation)
end_index	Integer	A position of the last -1 that is located in the "path" array (related to shortest path evaluation)
i	Integer	An integer acted as a "pointer" that scan through the "path" array (related to shortest path evaluation)
path	Array of integer	All direction that the robot moves from one cell to another stores in an array

Conclusion

This paper presents the maze-solving robot using the Robot C programming. The functionality of the mechanical structure of the robot and the coding algorithms are tested manually within the given maze. As with the amount of the different function runs, we were able to find the proper threshold for the sensor wall detection algorithm and the encoder value for the motor turns. However, the inconsistent position error caused by the backlash and the friction was unavoidable. We used the additional function to align the robot to the center of the cell which has delayed the robot to reach the final cell, but the performance of the robot was improved. Therefore, we can conclude that the developed functions for solving the maze are usable and the overall robot performance is acceptable.

Appendix

Source code

```
#pragma config(Sensor, S1,     wall1,          sensorEV3_Touch)
#pragma config(Sensor, S3,     wall2,          sensorEV3_Touch)
#pragma config(Sensor, S$4,     ultrasound,     sensorEV3_Ultrasonic)
#pragma config(Motor,  motorA,          rightwheel,    tmotorEV3_Large, PIDControl,
encoder)
#pragma config(Motor,  motorD,          leftwheel,     tmotorEV3_Large, PIDControl,
encoder)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//
//Gear ratio 24:16

//Create a structure representing each cell on the maze
typedef struct{
      int NWall;
      int SWall;
      int EWall;
      int WWall;
      char Visited;
}Cell;

Cell Maze[4][6]; // creates two dim. (4x6) array of structs

int encodeValueMotorA;
int encodeValueMotorD;
int Walldistance = 10;
int direction;
int soundActive = 0;
void cellinitial();
void ResetValue();
void Stop();
void TurnLeft();
void TurnRight();
void GoFwd();
void Adjust();


// direction
// 0 north
// 1 east
// 2 south
// 3 west

task main()
{
       cellinitial();
       ResetValue();

       //The robot begin by facing north direction
       direction = 0;

       //Array for storing robot's moving path
       int path[100];
       for(int i = 0; i < 100; i++){
              path[i] = -1;
       }
       //Index for Path array
       int ind = 0;

       int r_start = 0;
       int c_start = 0;

       //Enter starting cell
       while(getButtonPress(ENTER_BUTTON) != 1){
        displayCenteredBigTextLine(4, "starting cell");
     displayCenteredBigTextLine(6, "row %d", r_start);
```

```
      displayCenteredBigTextLine(8, "column %d", c_start);
            if(getButtonPress(UP_BUTTON)){
                r_start++;
            }
               else if(getButtonPress(DOWN_BUTTON)){
                      r_start--;
               }
               else if(getButtonPress(RIGHT_BUTTON)){
                      c_start++;
               }
               else if(getButtonPress(LEFT_BUTTON)){
                      c_start--;
               }

               wait1Msec(200);

        }

        eraseDisplay();
        wait1Msec(300);

        int r_current = r_start;
        int c_current = c_start;

        int r_target = 0;
        int c_target = 0;

        //Enter the target cell
        while(getButtonPress(ENTER_BUTTON) != 1){
         displayCenteredBigTextLine(4, "target cell");
      displayCenteredBigTextLine(6, "row %d", r_target);
      displayCenteredBigTextLine(8, "column %d", c_target);

               if(getButtonPress(UP_BUTTON)){
                      r_target++;
               }
               else if(getButtonPress(DOWN_BUTTON)){
                      r_target--;
               }
               else if(getButtonPress(RIGHT_BUTTON)){
                      c_target++;
               }
               else if(getButtonPress(LEFT_BUTTON)){
                      c_target--;
               }

               wait1Msec(200);
        }

        displayCenteredBigTextLine(4, "starting cell");
        displayCenteredBigTextLine(6, "row %d column  %d", r_current,c_current);
        displayCenteredBigTextLine(8, "target cell");
        displayCenteredBigTextLine(10, "row %d       column  %d", r_target,c_target);
        wait1Msec(500);

        flushButtonMessages();
        Stop();
        waitForButtonPress();
        eraseDisplay();

        //Start to solve the maze
         //Bumping into the wall to realign if there is a wall in the front & update wall
information
        if(SensorValue[S4] <= Walldistance){
               Maze[r_start][c_start].NWall = 1;
               Stop();
               Adjust();
        }
```

```
int adjustActive = 0;
int starting = 1;

while(r_current != r_target || c_current != c_target){
     //Update "visited" information
        Maze[r_current][c_current].Visited = 'Y';
        if(SensorValue[S4] <= Walldistance){
         //To activate the adjust fuction (for realignment)
         adjustActive = 1;
        }
        TurnRight();
        //If there is a wall
        if(SensorValue[S4] <= Walldistance){
         //Update the wall information to the structure called maze
                switch(direction){
                        case 0: Maze[r_current][c_current].NWall = 1; break;
                        case 1: Maze[r_current][c_current].EWall = 1; break;
                        case 2: Maze[r_current][c_current].SWall = 1; break;
                        case 3: Maze[r_current][c_current].WWall = 1; break;
                        default : break;
                }
                if(adjustActive == 1 || starting){
                        Stop();
                        Adjust();
                        starting = 0;
                }
                TurnLeft();
                while(SensorValue[S4] <= Walldistance){
                        switch(direction){
                                case 0: Maze[r_current][c_current].NWall = 1; break;
                                case 1: Maze[r_current][c_current].EWall = 1; break;
                                case 2: Maze[r_current][c_current].SWall = 1; break;
                                case 3: Maze[r_current][c_current].WWall = 1; break;
                                default : break;
                        }
                        if(adjustActive == 1 || starting){
                                Stop();
                                Adjust();
                        }
                        TurnLeft();
                }
                GoFwd();
         //Update the position of the robot
                switch(direction){
                        case 0: r_current++; break;
                        case 1: c_current++; break;
                        case 2: r_current--; break;
                        case 3: c_current--; break;
                        default : break;
                }
        }
        //If there is no wall
        else{
                GoFwd();
         //Update the position of the robot
                switch(direction){
                        case 0: r_current++; break;
                        case 1: c_current++; break;
                        case 2: r_current--; break;
                        case 3: c_current--; break;
                        default : break;
                }
        }
    //Store the path that robot move
        path[ind] = direction;
        ind++;
        adjustActive = 0;
```

```
                //Display position and direction on the brick screen
                displayCenteredBigTextLine(4, "row %d column  %d", r_current,c_current);
                displayCenteredBigTextLine(6, "direction %d", direction);

                //Activate the sound when the robot reach the target cell
            if( r_current == r_target && c_current == c_target){
                        soundActive = 1;
                }
        }
         //Make a beep for 2 sec. once the robot reach the target cell
        while(soundActive == 1){
                playTone(300,200);
                wait10Msec(200);
                soundActive = 0;
        }

        //Moving back to the starting cell using the shrotest cell
        //ind = the length of path array (from home to cheese)
        int begin_index;
        int end_index;
        int i = 0;

        //Scan through array to find shortest path
         //Scan through an array if there is a pair of opposite direction (N&S or E&W)
         //replace those elements with -1
        while(i < ind){
                if(path[i] == -1){
                        if(i == 0){
                                begin_index = i;
                        }
                        else{
                                begin_index = i-1;
                        }
                        while(path[i] == -1){
                                i++;
                        }
                        end_index = i;
                        if(abs(path[begin_index]-path[end_index]) == 2){
                                path[begin_index] = -1;
                                path[end_index] = -1;
                                i = 0;
                        }
                        else if(begin_index == 0 && abs(path[end_index]-path[end_index+1])
== 2){
                                path[end_index] = -1;
                                path[end_index+1] = -1;
                                i = 0;
                        }
                }
                else if(abs(path[i]-path[i+1]) == 2){
                        if(i == 0){
                                path[i] = -1;
                                path[i+1] = -1;
                                i = 0;
                        }
                        else if(i != 0 && path[i+1] != -1 && path[i-1] != -1 && i != 0){
                                path[i] = -1;
                                path[i+1] = -1;
                                i = 0;
                        }
                        else{
                                i++;
                        }
                }
                else{
                        i++;
                }
        }
```

```
        //Convert the direction & Go back to starting point
        i = 0;
        while(i < ind){
                switch(path[i]){
                        case 0: path[i] = 2; break;
                        case 1: path[i] = 3; break;
                        case 2: path[i] = 0; break;
                        case 3: path[i] = 1; break;
                        default : break;
                }
                i++;
        }

        for(int i = ind-1; i >= 0; i--){
                if(path[i] >= 0){
                        while(path[i] != direction){
                                ResetValue();
                                if(path[i] > direction && path[i]-direction != 3){
                                        TurnRight();
                                        Stop();
                                }
                                else{
                                        TurnLeft();
                                        Stop();
                                }
                        }
                        ResetValue();
                        GoFwd();
                        Stop();
        }
      }


}


//Functions:

//Initialize the outer wall of the maze & mark every cell as "unvisited"
void cellinitial(){
        for(int c=0;c<6;c++){
                        Maze[0][c].SWall=1; // Bottom row
                        Maze[3][c].NWall=1; // Top row
        }

        for(int r=0;r<4;r++){
                        Maze[r][0].WWall=1; // Left Col
                        Maze[r][5].EWall=1; // Right Col
        }
        for(int r=0;r<4;r++){
                for(int c=0;c<6;c++){
                        Maze[r][c].Visited = 'U';
                }
        }
}

//Reset the encoder value of 2 large motors
void ResetValue(){
    resetMotorEncoder(motorA);
     resetMotorEncoder(motorD);
     encodeValueMotorA = getMotorEncoder(motorA);
     encodeValueMotorD = getMotorEncoder(motorD);
}

//Stop the motors
void Stop(){
     motor[motorA] = 0;
     motor[motorD] = 0;
}
```

```
//Turn 90 degrees left and update the direction that robot is facing
void TurnLeft(){
      if(direction <= 0){
        direction = 3;
      }
      else{
        direction--;
      }
    ResetValue();
      while(encodeValueMotorD < 90*2.98 && encodeValueMotorA > -90*2.98){
         encodeValueMotorA = getMotorEncoder(motorA);
         encodeValueMotorD = getMotorEncoder(motorD);
         motor[motorA] = -10;
         motor[motorD] = 10;
      }
      Stop();
}

//Turn 90 degrees right and update the direction that robot is facing
void TurnLeft(){
void TurnRight(){
      if(direction >= 3){
        direction = 0;
      }
      else{
        direction++;
      }
    ResetValue();
    while(encodeValueMotorA < 90*3.02 && encodeValueMotorD > -90*3.02){
        encodeValueMotorA = getMotorEncoder(motorA);
         encodeValueMotorD = getMotorEncoder(motorD);
         motor[motorA] = 10;
         motor[motorD] = -10;
      }
      Stop();
}

//Goes forward from a center of one cell to a center of another cell
void GoFwd(){
    ResetValue();
      switch(direction){
         case 0:
             while(encodeValueMotorA < 725 && encodeValueMotorD < 730){
                 encodeValueMotorA = getMotorEncoder(motorA);
                 encodeValueMotorD = getMotorEncoder(motorD);
                 motor[motorA] = 40;
                 motor[motorD] = 40;
             }
             break;
         case 1:
             while(encodeValueMotorA < 745 && encodeValueMotorD < 750){
                 encodeValueMotorA = getMotorEncoder(motorA);
                 encodeValueMotorD = getMotorEncoder(motorD);
                 motor[motorA] = 40;
                 motor[motorD] = 40;
             }
             break;
         case 2:
             while(encodeValueMotorA < 725 && encodeValueMotorD < 730){
                 encodeValueMotorA = getMotorEncoder(motorA);
                 encodeValueMotorD = getMotorEncoder(motorD);
                 motor[motorA] = 40;
                 motor[motorD] = 40;
             }
             break;
         case 3:
             while(encodeValueMotorA < 745 && encodeValueMotorD < 750){
                 encodeValueMotorA = getMotorEncoder(motorA);
```

```
                encodeValueMotorD = getMotorEncoder(motorD);
                motor[motorA] = 40;
                motor[motorD] = 40;
            }
            break;
        default: break;
    }
    Stop();
}

//Get called for realignment when there are NWall & EWall or EWall & SWall or SWall &
WWall or WWall & NWall
//Bumping into the wall and back up to stay on the center of a cell
void Adjust(){
     while(SensorValue[S1] != 1 || SensorValue[S3] != 1){
       if(SensorValue[S1] == 1){
              motor[motorA] = 0;
            motor[motorD] = 50;
        }
        else if(SensorValue[S3] == 1){
            motor[motorA] = 50;
            motor[motorD] = 0;
        }
        else{
            motor[motorA] = 50;
            motor[motorD] = 50;
        }
    }
    Stop();
    wait1Msec(1000);
    switch(direction){
        case 0:
            while(SensorValue[S4] <= 4){
                motor[motorA] = -5;
                motor[motorD] = -5;
            }
            break;
        case 1:
            while(SensorValue[S4] <= 4.5){
                motor[motorA] = -5;
                motor[motorD] = -5;
            }
            break;
        case 2:
            while(SensorValue[S4] <= 4){
                motor[motorA] = -5;
                motor[motorD] = -5;
            }
            break;
        case 3:
            while(SensorValue[S4] <= 4.5){
                motor[motorA] = -5;
                motor[motorD] = -5;
            }
            break;
        default: break;
    }
    Stop();
}
```

Link to the video

https://youtu.be/aCVKrm_rBq4