# Lego Robot Project Report – Maze Explorer

## 1 Project Team

Steve Wanhella akira0@home.com
Tony(Laxmikanta Mishra) mishralk@hotmail.com
Wayne (Wenguang) Wang wew036@cs.usask.ca

## 2 Project Mission

Our mission is to let the Lego robot find a path to the target square on a maze. The maze consists of many equal-sized squares which are marked as Roads, Branches, Walls, and Goal by different colours. We built a Lego robot, named as maze explorer, which can find the path to the target square. The robot does not know the structure of the maze. It can start from any point of the maze except walls. It will go straight on the path until it reaches the obstacle. Then the Robot moves backward and chooses another path. It can try every possible path on a branch. In this way the Robot keeps moving until it finds the Goal. When it finds the Goal our mission is over.

## 3 Project Overview

Our program consists of the five modules (see Figure 1). The Control module resides in the RCX box which is a downloadable program. The other four modules resides on PC which is programmed in Visual Basic.

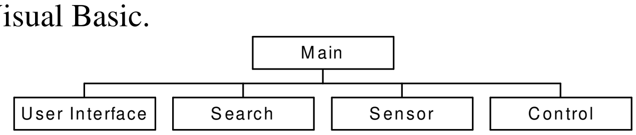



**Figure 1 System architecture**

After you start the program, you will visualise the main window (see Figure 1). This window has the options and capabilities to help the user in giving information about the current position, next movement, the direction of the Robot.

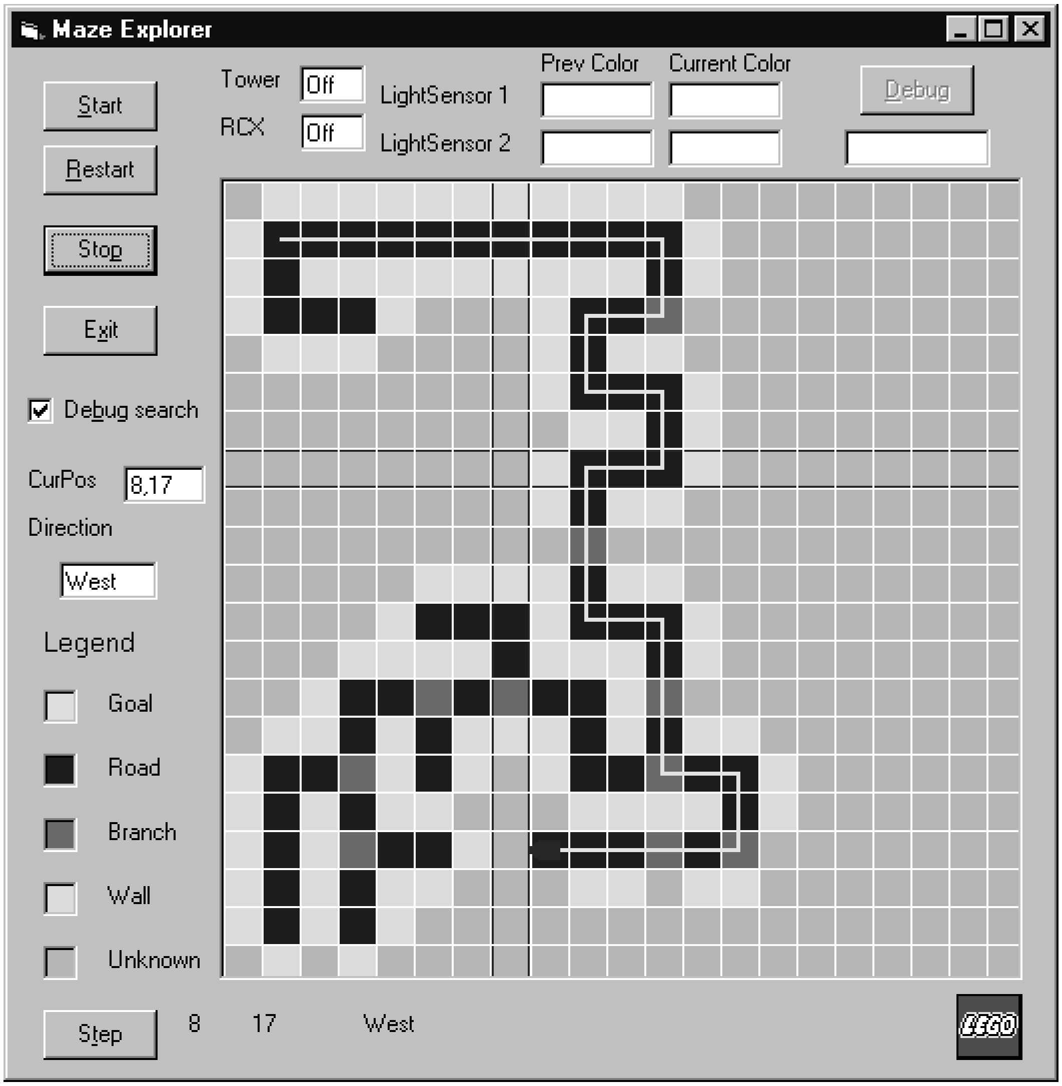



**Figure 2 Main Window**

# 4 User Interface

The User Interface enables you to view, examine, control and manipulate the program

## *4.1 Running Environments*

The Microsoft Visual Basic (VB98) must have been installed on the PC. The Robotics Invention System should also be installed. An infrared tower is connected to a serial port of the PC (the long term transmission is recommended). The RCX robot should be inside the infrared range of the tower.

## *4.2 Main Window*

The buttons on the top left are buttons for controlling the robot.

- Start**:** Pressing the Start button enables the Virtual Robot to move on the screen and the Real Robot on the maze.

- Stop**:** Pressing the Stop button enables the Virtual Robot to stop on the screen and the Real Robot on the maze.
- Restart: If the robot is stopped before, pressing the Restart Button enables the user to clear the map and start the robot again.
- Exit: Pressing the Exit button enables the user to exit the program.

The current information (location, direction) of the robot is shown on the left side.
- Direction: This Text field shows the Direction of the Robot on the Maze by indicating like, North , South, East and West).
- CurPos: The maze has several squares and they are designed in the Computer by defining in the state of Row and Column. This CurPos Text field show the currrent position of the Robot like, (1,1) or (R,C) where R stands for the Row and C stands for the Column.

The legends on the left side show the meaning of the different colours.
- Green: Branch of the Road (Where Robot can change the path).
- Black: Road (Path).
- Yellow: Goal (Piece of Gold).
- Any other colour: Obstacle on the Road(path).

The check box *Debug Search* is checked when debugging the search module only. In this mode, the program does not control the real robot but generate a random maze in PC and explore it. The buttons and texts on the bottom of the window show the information of this debug module.
- Step: This button is only used when the *Debug Search* is checked. It can debug the search module step-by-step.

The top of the window shows some information of the real robot.
- Tower: In this Text field we can visualise the current status of the Tower which controls the real Robot on the maze, The status *off* means either the Tower is not functioning , or is not connected to the Computer. The status *on* means the tower is ready to work. To check the truth you can check the green light of the tower.

- RCX: The RCX Text field shows the current status of the RCX. RCX is the Brick or you can think it as the Engine of the Robot in which we find the motors, memories etc. (you can find the details in the Lego Mindstorm Manual). If it shows the status to be *on* that means the Robot is ready to work and the opposite is true if the status is shown as *off*.

- Light Sensor: This text field shows the current position of the Robot, for example if the Robot is on the Black square, this Text field will show you the square as Road which implies to be the colour of the Road(Path).

### 4.3 Debug Window

Pressing the *Debug* button on the top right of the main window can pop up the debug window for the sensor and control module (see Figure 3). All functions of the control and sensor modules can be tested here. The functions of most buttons can be deduced from their names.

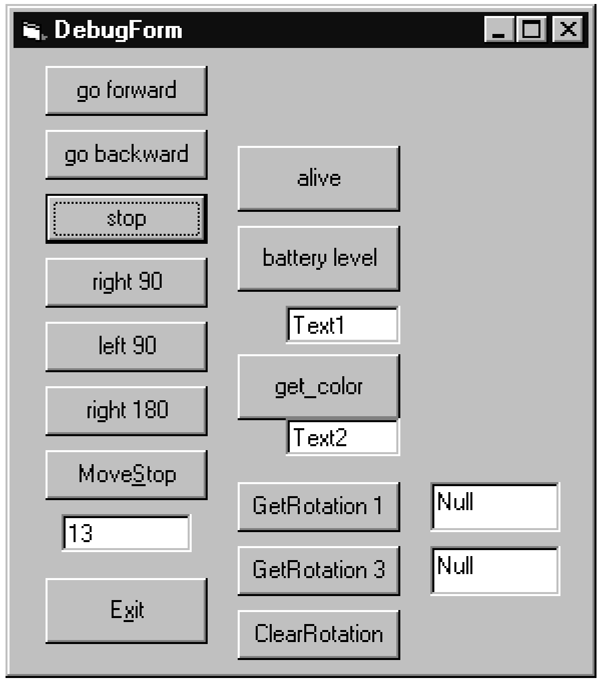



**Figure 3 Debug Window**

- Alive : Pressing this button enables to know about the current condition of the RCX and the Tower.
- Batterylevel: Pressing this button will enable the use to know about the current status of the battery, which means you need new batteries or not.
- PrevColor: This Text field shows the previous colour, which is actually the previous square of the maze ( might be a Road, an Obstacle or a Branch).
- CurrentColor: This Text field shows the current colour, which is actually the present square of the maze on which the Robot is right now ( might be a Road, an Obstacle, or a Branch).

## 5 Control Module

### 5.1 Introduction

The control module is meant for controlling the Robot while searching for the Goal on the Maze. While doing that, this module performs two basic functions as follows ;

1. MoveAndStop().
2. Rotate().

### 5.2 Objectives

**<u>Move and Stop():</u>**

This function is defined to enable the Robot to move straight for a certain distance, where distance is declared as an integer value in this module. There are two motors in the Brick named 'motor 0' and 'motor 2'. In the initMoveandStop sub routine. Distance values are declared for each rotation sensor. For move and stop, each distance is the same for each sensor. If that distance happens to be negative, then that particular motor is set to run in rwd. This is very useful because this allows the code to be reused for the rotate function.

Move and stop along with rotate are downloaded to the RCX at the time of initialization. When we used this function with communication over the IR tower, the delay was devastating. For a move and stop of say 13 we would get an actual value of anywhere between 15 and 20! For this reason MoveandStop is contained internally and seems to work very well there. For greater accuracy, there are variables R0extra and R2extra whose value gets calculated after a move. If the actual value differs from the desired value, then the difference is stored in R*extra. On the next move this will either be added or subtracted to the distance. This system has worked for us with some success.

**<u>Rotate()</u>**

The rotate function is really exactly the same as MoveAndStop but with a different setup function for setting what is distance. Rotation is called with a certain angular value; 90, -90, or 180. To set the distance we check which one: if 90 then set distance of the right rotation sensor to 11, and the distance of the left rotation sensor to –11. The same is oppositely true for –90. 180 then is simply 22 (which one to set negative depends on desired direction of turn which really doesnt matter).

Rotate also must be inside the RCX for the same reason as MoveAndStop. Using the IR tower, it was impossible to stop the robot at 90 degrees, or even close to it unless by sheer luck. Contained within the robot it is much more reliable. The big problem we still have with rotation is not completely with the sensors but also with the slipping treads.

# 6 Sensor Module

The sensor module contains all of the code that deal with the sensors. The sensors for this robot are, one light sensor, and two rotation sensors. The sensor module contains the following essential functions:

1. Initialize sensors
2. Clear Rotation sensors
3. Get Color (sensor)
4. Read Rotation sensor (sensor)

<u>1. Initialize </u>sensors sets the light sensor as follows:

```
SetSensorType Sensor_2,  light_type
        SetSensorMode Sensor_2, Raw, 0
SetSensorType Sensor_1,  Angle (7)
        SetSensorMode Sensor_1, 3, 0
SetSensorType Sensor_3,  Angle (7)
        SetSensorMode Sensor_3, 3, 0
```

Sensor mode was strage for the rotation sensors. Only 3 would work for us which is the predefined value for a reflection sensor.

<u>2. Clear</u> rotation sensors simply clears the value of both sensors, setting them to 0:

```
ClearSensorValue(Sensor_1)
ClearSensorValue(Sensor_3)
```

<u>3. Get Color</u> uses defined color values to determine colors. These are manually read from Rcx. They are stored as static public variables in "Globals.bas". These values are used with a Variance variable to determine color:

```
X = poll(Sensor_2, senval)

if (X - Variance) < Gold  < (X + Variance)
        then get_color = "goal"
if (X - Variance) < Blue < (X + Variance)
        then get_color = "road"
etc
else get_color = "wall"
```

<u>4. Read Ro</u>tation sensors takes the sensor name as input. It simply reads and returns the sensor value.

```
ReadRotation = poll(SENVAL, sensor)
```

# 7 Search module

Search Module makes decisions on where and when to move the robot based on certain search strategy and the input of sensors. Because the information of the maze is incomplete and the robot cannot jump from one square to a remote square, the simple recursive algorithm cannot be used here. However, a similar idea is used. A stack is used to store the passed path of the robot.

The maze is composed of four kinds of squares: Road, Branch, Wall, and Goal. For a Road square, there are at most two adjacent Road/Branch/Goal squares. For a Branch square, there can be up to four adjacent Road/Branch/Goal squares.

The basic strategies are:

- Use depth-first search.

- If the robot reaches a square that it has reached before, it will go back to avoid dead loop. Therefore, the robot works even if the maze contains loops.
- Go straight unless hit a wall.
- Remember all passed routes by using a stack.
- Deduce the location of walls based on the property of the Road square. In Figure 1, if the robot goes from A through B to C, and B is a road, then robot knows that both D and E are walls. It won't go into D or E in the future.

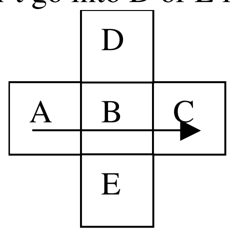



**Figure 1 Deduce the locations of walls**

- When all squares ahead are tested, the robot will go back along the coming route, and try another branch at a branch square.

# 8 Test Cases

In these test drawings the colors are defined as follows:

Gold – this is the goal square of the maze.

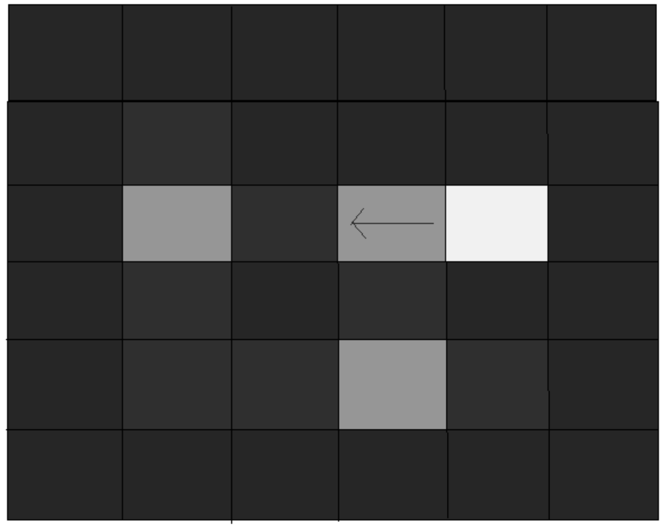
Wall – this is also the default color of the background

Road – A road can have at most two adjacent traversable squares.

Branch – Up to four adjacent traversable squares.

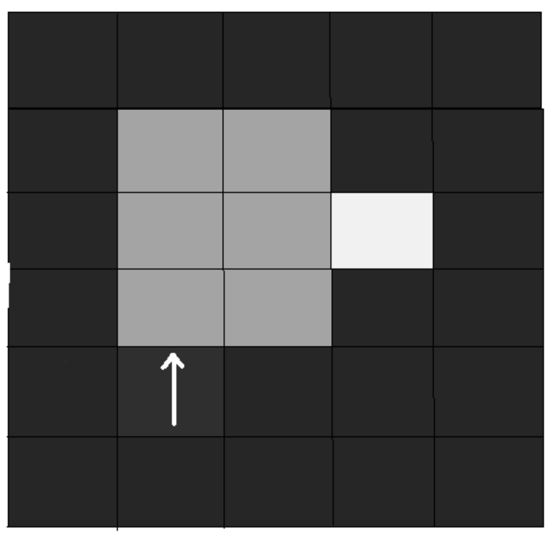
## *8.1 Test Case 1 (Normal)*

The first test case was designed to test a cycle as well as to practice the basic function of the robot. From the arrow (start point and direction) it worked its way around the circle until it reaches the initial square again. The old algorithm would then make the robot backtrack all the way back until it finally reached the gold. The new cycle algorithm has a special test for cycles. If they don't contain any unchecked branches then it can simply continue.



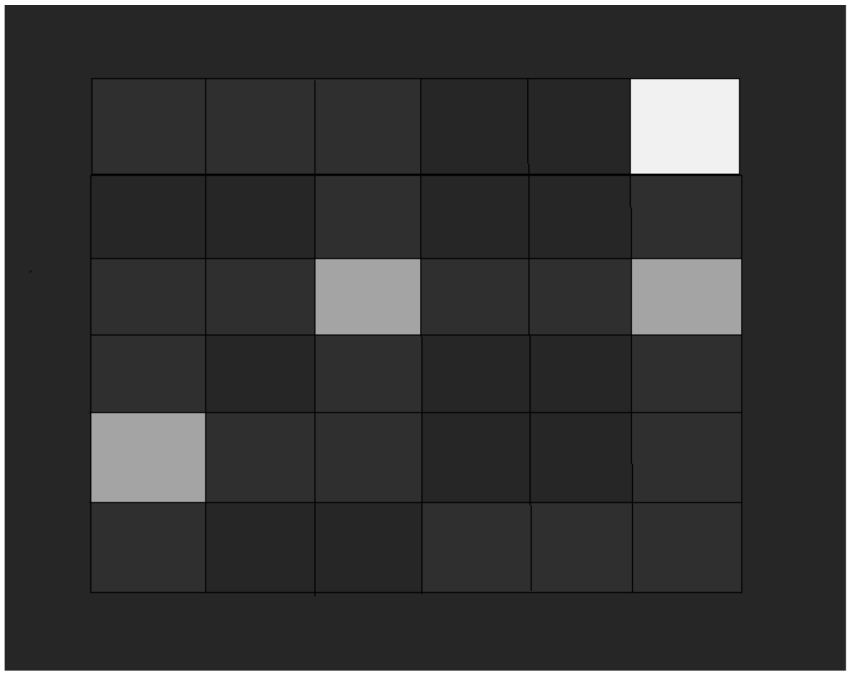
### 8.2 Test Case 2 (Many Branches)

This was a versatility test case with a bizarre but legal map. This map simulates a large open area. As you would expect however the robot simply works its way all the way around and then starts checking branches finding the goal easily.



### 8.3 Test Case 3 (Biggest)

This is our final design. It contains a cycle for interest and is intended to have no predefined start square. The robot may be placed in any square and facing any direction and it will find the goal.



## 9 Technical Manual (Lego Building)

### 9.1 The Maze

The maze is constructed on a piece of wood (masonite) and the tiles of the maze are different colored pieces (6" X 6") of construction paper. Different colors are used for the road, goal, and branch. The walls are the parts of the maze that aren't represented by squares. Thus the brown color of the masonite is the wall.

### 9.2 Color Considerations

The maze is constructed with different colors denoting different square types. The square types for our maze were:

1. Goal (gold)
2. Road (black)
3. Branch (green)
4. Wall (other / brown)

These colors were chosen for their distinction. The light sensors are not always very accurate. They may be affected by shadows, marks on the paper, and distance from the surface. These colors allowed us to have a Variance (plus or minus) of up to 15, which enhanced the recognition skills of the robot to an acceptable level. An interesting idea brought up in class was the use of aluminum foil to provide a unique return value. Our use of OTHER for the wall color allowed up to simply use the height from the edge of the board to act as a wall.

### 9.3 Robot Design

We stuck pretty close to the suggested design of the robot. Our sensor configuration is one light sensor, and two rotation sensors. The light sensor is mounted on the front-center with a long arm to place the sensor in the middle of the next square, and the rotation sensors are connected each to a gear on either side. We used a the largest gear to connect to the smaller gear of the motor which then drives a medium sized gear on the wheels. It was important for the maze robot to go slowly for more control, although a super fast maze robot would be very entertaining!

Our robot uses treads as in the pictured robot on the web. We had and still have problems with these treads. It is very difficult to turn exactly 90 degrees and stay in one place. Usually either the treads slip so much that 90 degrees is never reached or exceeded, or it turns 90 and floats off to the right or left. The other groups had trouble with wheels as well so we stuck with treads. One group (Chad and Nathan) claim to have overcome this problem by using a differential.

It would be desirable to have the light sensor mounted on a spring operated wheel leg to maintain distance from the paper. We found that securely taping down the squares was sufficient.

## 10 Problems and Experiences

### 10.1 Debugging and Testing of the Search Module

The logic of the search module is very complex. The debugging of it is not easy. By using a DebugSearch module, we solved this problem successfully.
This DebugSearch module can simulate the lower layer (Control and Sensor modules) completely in PC. Based on this DebugSearch module, the search module can run without the real Lego robot. This facilitates the debugging and testing of the Search Module.
A maze generator is also programmed in the DebugSearch module. This maze generator can generate variable sized random mazes. After the search module is tested on some very big, complex mazes generated by the maze generator, we are sure that the logic of the search module is right.

### 10.2 Inaccuracy of the Robot Control and Sensor

Our robot uses light sensors and rotation sensors to sense the world — the map of a maze. The light sensor is used to read the color of the map, and the rotation sensor can let the robot know where it is. Both sensors are not very accurate. However, we managed to make them works pretty well at last.

### *10.3 Light Sensor*

There is one light sensor in our robot. It can detect different colors. However, the sensor value for the same color can change when the light of the environment changes, or the distance between the light sensor and the map changes.

The light sensor also cannot distinguish white, red, yellow. Therefore, we use dark, gray, yellow, and light green colors to represent the Road, Branch, Goal and Wall, respectively. We also define the Wall color as this: any color that is not Road, Branch, or Goal is Wall. Therefore, when the robot reaches the edge, it will also think a wall is ahead.

Because the value from the light sensor may fluctuate, we use a variance. Therefore, if the color value of Road is 820, all colors in [820-variance, 820+variance] are considered to be Road. Our tests show that the robot can recognize the map of the maze correctly.

To allow some errors during the movement, the robot always tries to stop and read the color at the center of the squares. Therefore, if the error distance is less than half of the square length, it can still get the correct color.

### *10.4 Control of the Movement*

The robot needs two basic movements: *MoveAndStop* moves the robot a certain distance and stops; *Rotate* let the robot rotate 90, -90, or 180 degrees.

Both the motors and the rotation sensors are needed to accomplish these two tasks. The basic idea is: start the motors, and then read the value of the rotation sensors until the distance is achieved, stop the motors at last.

#### 10.4.1 Transmission Delay Between PC and Robot

At the beginning, we put this logic into PC. However, because there is a big transmission delay between the PC and the robot, the robot often goes 3 or 4 distance units when the rotation sensor value is transmitted to PC and the control commands are sent back. Therefore, the accurate control through PC is impossible.

The solution is putting the lower control logic into the Robot. The control delay is much shorter and more stable.

Another point to avoid the control delay is that when two motors are both need to be stop, stop them in ONE command, not in two commands. Otherwise they may not stop at the same time.

#### 10.4.2 Dynamic Adjustment

Because the speeds of the two motors are not exactly the same, the value of the two rotation sensors can also be different. Because the distances in the maze are not very long, the adjustment during the running is unnecessary. There is only adjustment at the end of the running: when one rotation sensor reaches the distance, stop the motor which is on the same side of this rotation sensor. Therefore, if a motor goes faster, it will stop earlier, and both sides will go through the same distance. This idea applied to both the MoveAndStop and the Rotate.

One of our failed try was to adjust the direction of the robot at the edge of the square by using two light sensors. However, the response time and accuracy of light sensors is much worse than rotation sensors. We gave up this and changed to rotation sensors.

#### 10.4.3 Fine Control of the Move

When rotating the robot 90°, 11 clicks of rotation sensor is too less, and 12 is too much. Therefore, we let it use 11 and 12 alternately. After applying this technique, there is only very little error if we do the same rotation many times. However, there is still error on each turn, and it may accumulate because we do not let it turn in the same direction repeatedly.

#### 10.4.4 Avoidance of Error Accumulation

Even using the dynamic adjustment, the robot still can not run exactly the distance as expected. There is a minor error on every step. If there are no corresponding adjustments, the accumulation errors can change the result after many steps. Therefore, after each move, we record the minor error and correct it in the next move. We use the LeftExtra and RightExtra to record the extra distance the robot goes on last move. The algorithm is showed here:

```
Function Move(Distance as Integer) ' Move could be MoveAndStop or Rotate
        LeftRealDistance = Distance - LeftExtra
        RightRealDistance = Distance - RightExtra
        ……  ' Do the real move
        LeftExtra = LeftRotationSensorValue - (Distance - LeftExtra)
        RightExtra = RightRotationSensorValue - (Distance - RightExtra)
End Function
```

#### 10.4.5 Correcting Errors On-the-fly

In our experience, the avoidance of error accumulation still did not get the ideal result. Therefore, we changed the policy again. Instead of correcting the error on the next move, we correct the error after each move.

#### 10.4.6 The Delay After the Move

Because there is a long arm in our robot, the arm vibrates during the running. This fluctuation of the arm changes the distance between the light sensor and the surface. Therefore, after each move, we wait 500ms. After that, the arm is stable, and the correct color can be read from the light sensor.

### *10.5 Robot type problems*

The slipping treads are without a doubt our largest problem. It seems almost impossible with treads to make a perfect turn. If the rotation sensors were mounted externally to the robot, like Mikes robot then maybe this would not be as much of a problem. Ideally the maze robot would dwell in a maze with physical walls and use touch sensors and light sensors exclusively. Then it could fudge all of its turns and plow its way to the goal.

Another problem with the rotation sensor was the significance of the reading. One click of the rotation sensor is more than a degree, so to make a perfect 90 degree turn is almost impossible. We used the mod function to try to alternate and simulate a half a degree with successive turns. This was not altogether successful.

Another problem we had was working with two light sensors. We originally thought this would be a great way to straiten out the robot. When one side hits a new color, its motor stops and the other continues until it reaches the same color. The problem we had with this is that one light sensor read 35 higher than the other one. This is no big problem because it was consistent. Combined however with the fact that both of their readings fluctuate, and added with the delay in the IR tower to the Robot, it created a very unreliable maze robot. With only one light sensor, we move the robot guided by the rotation sensors into the middle of each square, read the color and then continue. This has proved to be much more reliable and predictable.