Maze solving using a robot

A Project report submitted in partial fulfillment of the requirements for the award

of the Degree of

Bachelor of Technology In

Computer Science and Engineering

By

B. Rahul Raju **Roll No: 10011P0502**

P. Phaneendra **Roll No: 10011P0514**

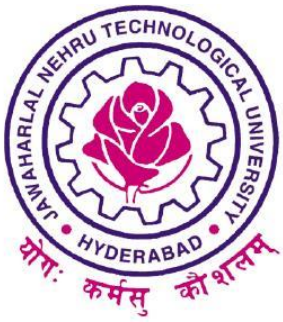
Raghav Mishra **Roll No: 10011P0517**

T. Siddartha Reddy **Roll No: 10011P0518**

Under the esteemed guidance of

Mrs. K Neeraja

Assistant Professor



Department of Computer Science and Engineering,

Jawaharlal Nehru Technological University Hyderabad

College of Engineering Hyderabad,

Kukatpally, Hyderabad-500 085.

Department of Computer Science and Engineering,

Jawaharlal Nehru Technological University Hyderabad

Kukatpally, Hyderabad-500 085.

DECLARATION BY THE CANDIDATE

We, **B. Rahul Raju (10011P0502), P. Phaneendra (10011P0514), Raghav Mishra (10011P0517) and T. Siddartha Reddy (10011P0518),** hereby declare that the project report entitled **"Maze solving using a robot",** carried out under the guidance of **Mrs. K Neeraja** is submitted in partial fulfillment of the requirements for the award of *Bachelor of Technology in Computer Science and Engineering*. This is a record of bonafide work carried out by us and the results embodied in this project have not been reproduced/copied from any source.

The results embodied in this project report have not been submitted to any other University or Institute for the award of any other degree or diploma.

B. Rahul Raju Roll No: 10011P0502

P. Phaneendra Roll No: 10011P0514

Raghav Mishra Roll No: 10011P0517

T. Siddartha Reddy Roll No: 10011P0518

Department of Computer Science & Engineering,

JNTUH college of Engineering,Hyderabad-500085.

Department of Computer Science and Engineering,

Jawaharlal Nehru Technological University Hyderabad

College of Engineering Hyderabad,

Kukatpally, Hyderabad-500 085.

CERTIFICATE BY THE SUPERVISOR

This is to certify that the project report entitled **"Maze solving using a robot",** being submitted by **B. Rahul Raju (10011P0502), P. Phaneendra (10011P0514), Raghav Mishra (10011P0517) and T. Siddartha Reddy (10011P0518)** in the Department of Computer Science And Engineering of **JNTUH COLLEGE OF ENGINEERING HYDERABAD** is a record of bonafide work carried out by them under my guidance and supervision. The results embodied in this project report have not been submitted to any other university or institute for the award of any degree or diploma. The results have been verified and found to be satisfactory.

Mrs K.Neeraja

Assistant Professor,

Department of Computer Science & Engineering,

JNTUH college of Engineering,

Hyderabad

Department of Computer Science and Engineering,

Jawaharlal Nehru Technological University Hyderabad

College of Engineering Hyderabad,

Kukatpally, Hyderabad-500 085.

CERTIFICATE BY THE HEAD OF THE DEPARTMENT

This is to certify that mini project report entitled "**Maze solving using a robot**" that is being submitted by **B. Rahul Raju (10011P0502), P. Phaneendra (10011P0514), Raghav Mishra (10011P0517) and T. Siddartha Reddy (10011P0518)** in the partial fulfillment of requirements for award of *Bachelor of Technology in Computer Science and Engineering*.

Dr. B. Padmaja Rani,
Professor & Head of the Department,
Department of Computer Science & Engineering,
JNTUH college of Engineering,
Hyderabad.

ACKNOWLEDGEMENTS

We owe a deep sense of gratitude to our guide, **Mrs K.Neeraja**, Assistant Professor, Department of Computer Science & Engineering, JNTUH College of Engineering, for her constant supervision, personal interest, critical evaluation and inspiring guidance in shaping this work.

In this opportunity we would like to express my sincere gratitude to our beloved Head of the Department**, Dr. B. Padmaja Rani,** Department of Computer Science & Engineering, JNTUH college of Engineering, for giving support to do this project.

Finally, we express thanks to all faculty members who have helped in successfully completing this project. Furthermore, we would like to thank my family and friends for their moral support and encouragement in completing the project.

B. Rahul Raju **Roll No: 10011P0502**

P. Phaneendra **Roll No: 10011P0514**

Raghav Mishra **Roll No: 10011P0517**

T. Siddartha Reddy **Roll No: 10011P0518**

Abstract

Maze Solving Using A Robot:

One of the most common and important aspect for a robot is self learning and decision taking. Robots of this kind have wide range of practical applications in various domains. Even though a robot can be presumed as a mechanical device or machine, its life lies in the program it is supposed to implement. The robot is programmed such that the program exploits the hardware of the robot to get the desired information. This information is processed and decision making is done based on this information.

The maze is set of dark lines on a bright background. The robot follows the dark lines using its Reflectance Sensors and decisions are taken based on this information. The Maze is solved using a simple algorithm **"Left Hand On The Wall",** where the takes only left turns at the intersections encountered. If any dead end is encountered, it takes a "U-turn". At every intersection encountered, the robot remembers the turn taken and maze is solved. All this is done in the first run of the robot through the maze. Now in subsequent runs , the robot directly reaches the destination in a definite path avoiding unnecessary turns. Hence the Maze is solved.

The robot used in this problem is a Pololu's 3pi robot especially designed for such problems. The Robot is programmed such that it intelligently learns the Maze and takes wise decisions on turnings.

Configuration Of the Robot:

Name: 3pi robot

Micro Processor: AT mega 328p (8-bit processor running at 20MHZ)

With 32KB flash Memory, 2KB RAM, 1KB EEPROM

Sensors: Five IR Reflectance sensors.

Power: Regulated 9.5 V from 4 AAA Batteries.

Software used:

IDE: Atmel Studio 6.1

Operating System: Windows 8

C/C++ Library: AVR Library.

CONTENTS

List of Figures

Chapter 1 – INTRODUCTION

1.1 Robotics:

Robotics is the branch of technology that deals with the design, construction, operation, and application of robots, as well as computer systems for their control, sensory feedback, and information processing . These technologies deal with automated machines that can take the place of humans in dangerous environments or manufacturing processes, or resemble humans in appearance, behavior, and/or cognition. Many of today's robots are inspired by nature contributing to the field of bio-inspired robotics. Robots have replaced humans in the assistance of performing those repetitive and dangerous tasks which humans prefer not to do, or are unable to do due to size limitations, or even those such as in outer space or at the bottom of the sea where humans could not survive the extreme environments.

A robot is a mechanical or virtual artificial agent, usually an electro-mechanical machine that is guided by a computer program or electronic circuitry. There is no consensus on which machines qualify as robots but there is general agreement among experts, and the public, that robots tend to do some or all of the following: move around, operate a mechanical limb, sense and manipulate their environment, and exhibit intelligent behavior — especially behavior which mimics humans or other animals. In practical terms, "robot" usually refers to a machine which can be electronically programmed to carry out a variety of physical tasks or actions.Robots have a very wide range of uses and scientists from decades have been working on giving the robots their own intelligence so that they could perform tasks in an automated manner. One of the very first kinds of robots is the Maze solving robots where the robots use the programming

intelligence given to them ,to solve a maze i.e, the robots find the correct feasible path from the source to the destination of a maze using the program code feeded to them.

1.2 MOTIVATION:

Research in robotic maze solving problems has been highly active for the past few decades in both robotics and artificial intelligence. The general objective for this kind of problem is to use a mobile robot to navigate an unknown or semi-known environment and to accomplish some specific tasks. Generally the common task is to navigate from the starting point to the goal or the destination point of the maze. A robot can be used in the situations where the path from the source to the destination cannot be perceived easily and where probability of finding the best feasible path is less.

1.3 PROBLEM DEFINITION:

In this project we address the problem of solving a maze using a robot. The maze has a source or a starting point with several routes branching out from it. There is a destination or ending point which is connected through several routes and not all the routes from the source connect to the destination point. There maybe several dead-ends in the maze and also many unfeasible paths from the source to the destination. A maze solving robot is used to find the feasible path from the source to the destination with the help of the programming intelligence given to it. In this project a Pololu 3pi robot is used to solve the maze.

1.4 LITERATURE SURVEY:

In this section the details of the chapters that are going to be discussed are explained in brief.

Chapter 2 –Pololu's 3pi Robot:

This chapter deals with the software and the hardware components that are used in the project. The hardware components constitute the robot and its parts and the software constitutes the code that is given to the robot in order to analyse the feasible path in the maze. Also the basic details related to those hardware and software components is discussed.

Chapter 3 – Maze solving robot

In this section, the strategy that is implemented to solve the maze is discussed. The study is made on how the robot identifies the path, the dead ends and also the destination using its hardware components. Finally the algorithm the robot implements to solve the maze is discussed.

Chapter 4 – Implementation

In Implementation, a detailed report is given on how the algorithm is implemented as code. The software that is rendered to the maze solving robot is thoroughly discussed in this section.

Chapter 5 – Experimental results

In this section the components , implementation and the output of the project is represented with the help of screen shots.

Chapter 6 – Conclusion and future work

In this chapter, a detailed report is given on what the project has achieved and how this project can be extrapolated further in order to solve more difficult and advanced problems.

Chapter 7 – References : Here all the references that are useful for the successful completion of the project are given a mention.

CHAPTER 2 - POLOLU's 3pi ROBOT

The Pololu 3pi robot is a small, high-performance, autonomous robot designed to excel in line-following and linemaze-solving competitions. Powered by four AAA batteries (not included) and a unique power system that runs the motors at a regulated 9.25 V, 3pi is capable of speeds up to 100 cm/second while making precise turns and spins that don't vary with the battery voltage. This results in highly consistent and repeatable performance of well-tuned code even as the batteries run low. The robot consists of two micro metal gear motors, five reflectance sensors, an 8×2 character LCD, a buzzer, three user pushbuttons, and more, all connected to a user-programmable AVR microcontroller.

The 3pi is based on an ATmega168 or ATmega328 micro controller .The ATMega328p micro processors are developed and manufactured by Atmel company. ATmega328p microcontroller, henceforth referred to as the"ATmegaxx8", runs at 20 MHz. ATmega328 feature 32 KB of flash program memory, 2 KB RAM, and 1 KB of persistent EEPROM memory.

2.1 *Why 3pi robot?*

The pololu's 3pi robot is a small and efficient robot designed especially for addressing line following and maze solving problems. The five reflectance sensors located beneath the robot makes it a great device for these kind of problems.

2.2. Power management:

Battery voltage drops as the batteries are used up, but many electrical components require a specific voltage. A special kind of component called a *voltage regulator* helps out by converting the battery voltage to a constant, specified voltage. For a long time, 5 V has been the most

common regulated voltage used in digital electronics; this is also called TTL level. The microcontroller and most of the circuitry in the 3pi operate at 5 V, so voltage regulation is essential. There are two basic types of voltage regulators:

• **Linear** regulators use a simple feedback circuit to vary how much energy is passed through and how much is discarded. The regulator produces a lower output voltage by dumping unneeded energy. This wasteful, inefficient approach makes linear regulators poor choices for applications that have a large difference between the input and output voltages, or for applications that require a lot of current. For example, 15 V batteries regulated down to 5 V with a linear regulator will lose two-thirds of their energy in the linear regulator. This energy becomes heat, so linear regulators often need large heat sinks, and they generally don't work well with high-power applications.

• **Switching** regulators turn power on and off at a high frequency, filtering the output to produce a stable supply at the desired voltage. By carefully redirecting the flow of electricity, switching regulators can be much more efficient than linear regulators, especially for high-current applications and large changes in voltage. Also, switching regulators can convert low voltages into higher voltages! A key component of a switching regulator is the *inductor*, which stores energy and smooths out current; on the 3pi, the inductor is the gray block near the ball caster labeled "100". A desktop computer power supply also uses switching regulators: peek through the vent in the back of your computer and look for a donut-shaped piece with a coil of thick copper wire wrapped around it – that's the inductor.

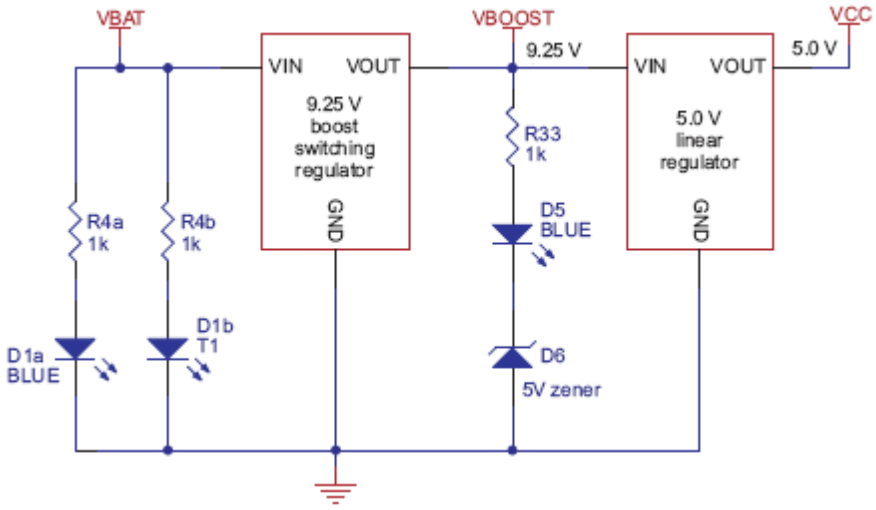


Fig .2.2 Block diagram of the Power management system of the 3pi robot.

2.3 Motors and Gearboxes:

A motor is a machine that converts electrical energy to motion. *A brushed DC motor* is used on the 3pi. A brushed DC motor typically has permanent magnets on the outside and several electromagnetic coils mounted on the motor shaft (armature). The primary values that describe a running motor are its speed, measured in rpm, and its torque, measured in kg·cm or oz·in (pronounced "ounceinches"). Multiplying the torque and speed (measured at the same time) give us the power delivered by a motor. We see, therefore, that a motor with twice the speed and half the torque as another has the same power output. Every motor has a maximum speed (when no force is applied) and a maximum torque (when the motor is completely stopped). We call these the *free-running speed* and the *stall torque*.

Driving a motor with speed and direction control

One nice thing about a DC motor is that you can change the direction of rotation by switching the polarity of the applied voltage. If you have a loose battery and motor, you can see this for yourself by making connections one way and then turning the battery around to make the motor spin in reverse. Of course, you don't want take the batteries out of your 3pi and reverse them every time it needs to back up – instead, a special arrangement of four switches, called an H-bridge, allows the motor to spin either backwards or forwards. Here is a diagram that shows how the H-bridge works:

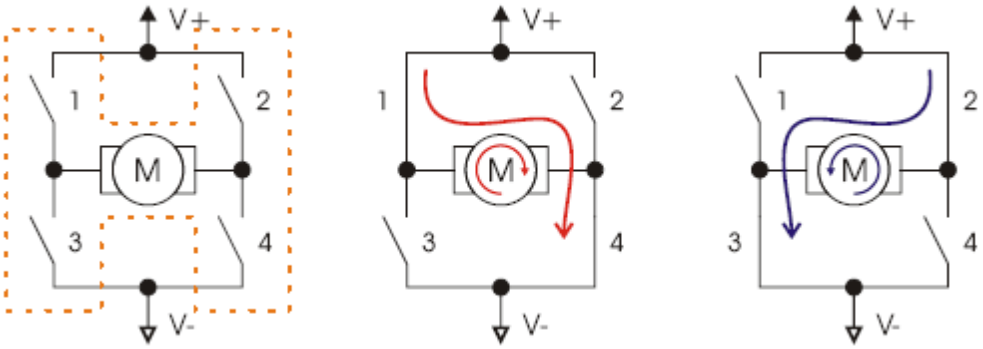


Fig.2.3 H-Bridge Of the motors and gear boxes

If switches 1 and 4 are closed (the center picture), current flows through the motor from left to right, and the motor spins forward. Closing switches 2 and 3 causes the current to reverse direction and the motor to spin backward. An H-bridge can be constructed with mechanical switches, but most robots, including the 3pi, use transistors to switch the current electronically. The H-bridges for both motors on the 3pi are all built into a single motor driver chip, the

TB6612FNG, and output ports of the main microcontroller operate the switches through this chip.

2.4 Digital inputs and sensors:

The microcontroller at the heart of the 3pi, an Atmel AVR mega168 or mega328, has a number of pins which can be configured as digital inputs: they are read by your program as a 1 or a 0 depending on whether the voltage is high (above about 2 V) or low.

Normally, the *pull-up resistor* R (20-50 k) brings the voltage on the input pin to 5 V, so it reads as a 1, but pressing the button connects the input to ground (0 V) through a 1 k resistor, which is much lower than the value of R. This brings the input voltage very close to 0 V, so the pin reads as a 0. Without the pull-up resistor, the input would be "floating" when the button is not pressed, and the value read could be affected by residual voltage on the line, interference from nearby electrical signals, or even distant lightning. Don't leave an input floating unless you have a good reason. Since the pull-up resistors are important, they are included within the AVR – the resistor R in the picture represents this internal pull-up, not a discrete part on the 3pi circuit board.

A more complicated use for the digital inputs is in the reflectance sensors. Here is the circuit for the 3pi's leftmost reflectance sensor, which is connected to pin PC0:

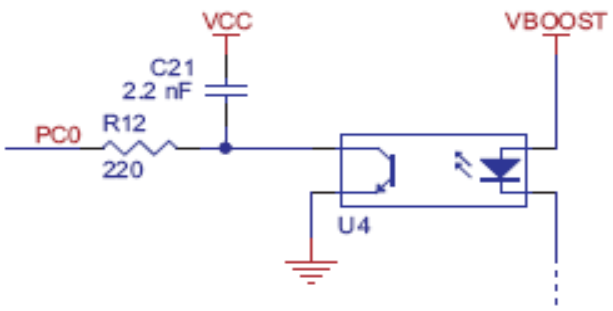


Fig.2.4 Circuit of 3pi's left most reflectance sensor on pin PC0

The sensing element of the reflectance sensor is the phototransistor shown in the left half of U4, which is connected in series with capacitor C21. A separate connection leads through resistor R12 to pin PC0. This circuit takes advantage of the fact the digital inputs of the AVR can be reconfigured as digital outputs on the fly. A digital output presents a voltage of 5 V or 0 V, depending on whether it is set to a 1 or a 0 by your program. The way it works is that the pin is alternately set to a 5 V output and then a digital input. The capacitor stores charge temporarily, so that the input reads as a 1 until most of the stored charge has flowed through the phototransistor. Here is an oscilloscope trace showing the voltage on the capacitor (yellow) dropping as its charge flows through the phototransistor, and the resulting digital input value of pin PC0 (blue):

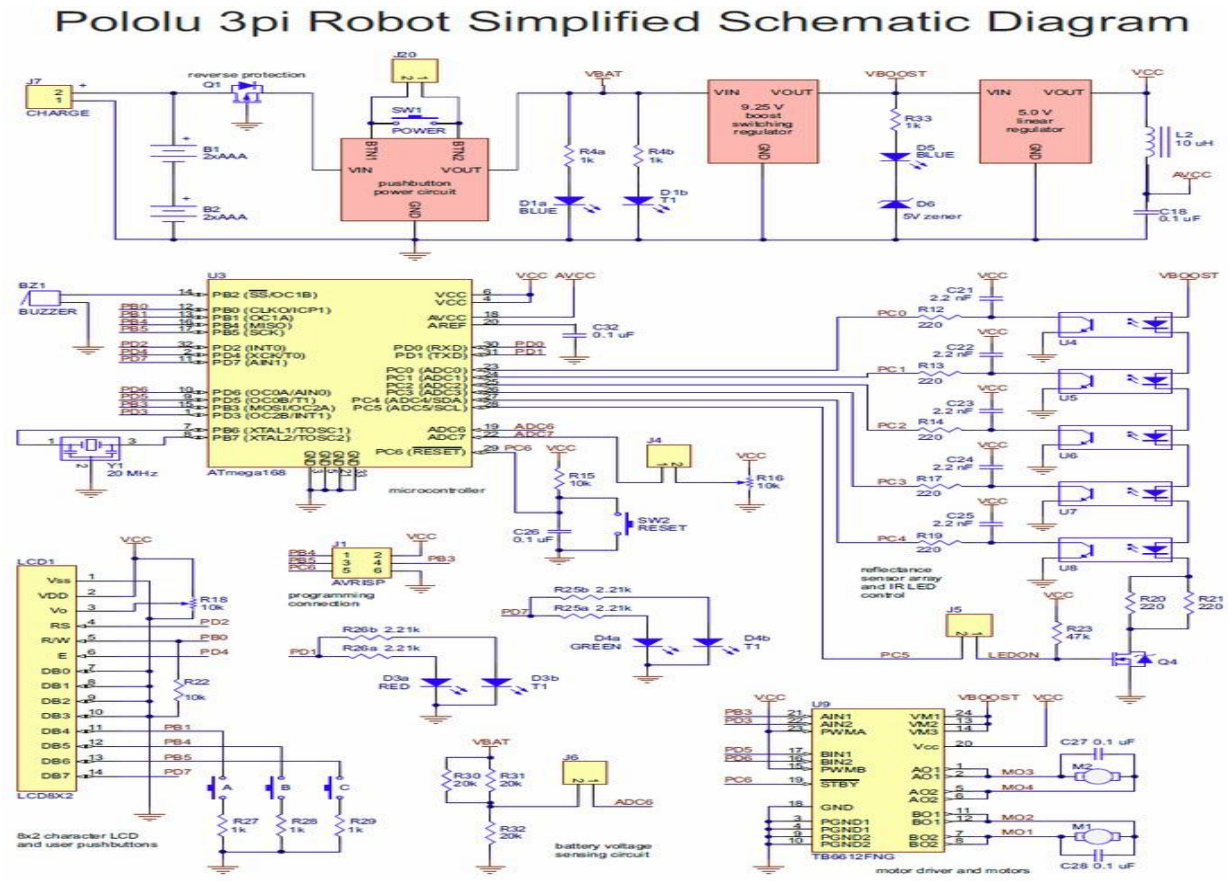
2.5 Pololu AVR USB Programmer:

The Pololu USB AVR programmer is a programmer for Atmel's AVR microcontrollers and controller boards based on these MCUs, such as Pololu Orangutan robot controllers and the 3pi robot. The programmer emulates an STK500 on a virtual serial port, making it compatible with standard AVR programming software. Two additional features help with building and debugging

projects: a TTL-level serial port for general-purpose communication and a SLO-scope for monitoring signals and voltage levels.

The Pololu USB AVR programmer connects to a computer's USB port via an included USB A to mini-B cable, and it connects to the target device via an included 6-pin ISP programming cable (the older, 10-pin ISP connections are not directly supported, but it is easy to create or purchase a 6-pin-to-10-pin ISP adapter).

2.6 3pi Simplified Schematic Diagram: The full understanding of the 3pi robot hardware can be understood by the following schematic diagram:



2.7 3pi Robot Machine Design:

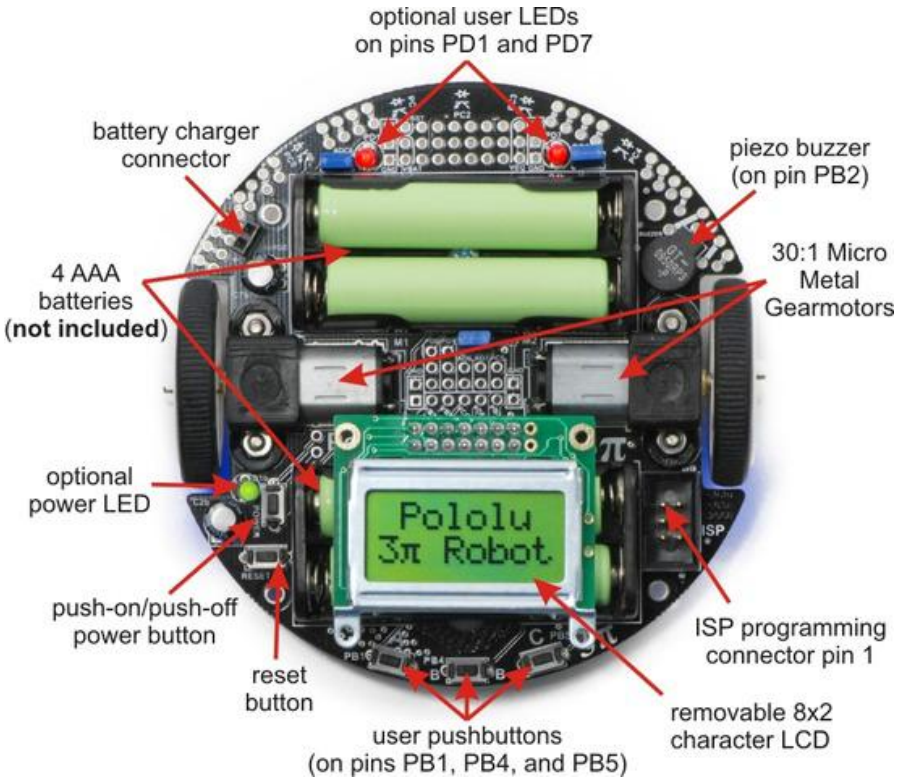


Fig.2.7.1 3pi robot's top view with its hardware components

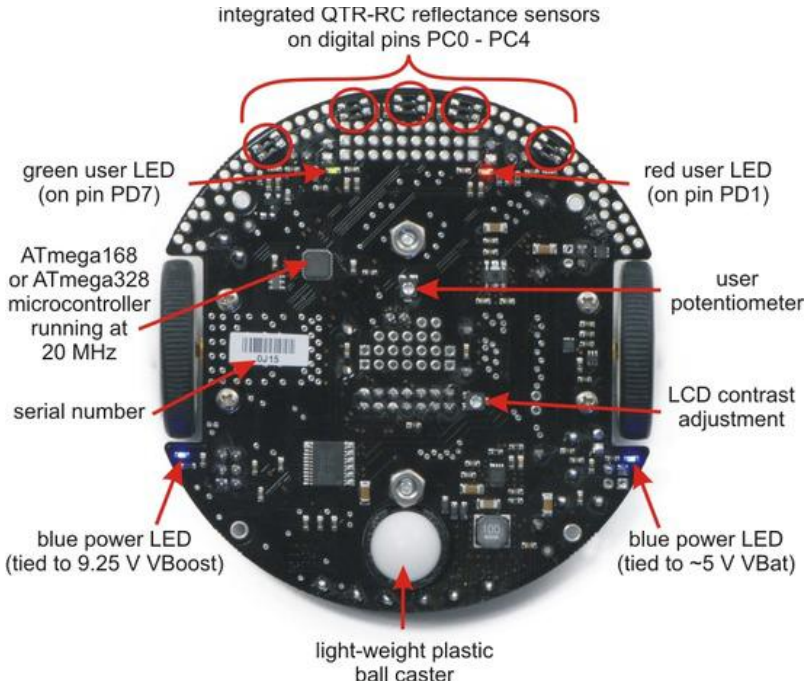


Fig.2.7.2 Pololu's 3pi robot bottom view with hardware and pins

2.8 Software Required:

The maze solving robot requires to be programmed. The program is written in an IDE by Atmel namely, **Atmel Studio 6.1**. The Program is written in C/C++ using AVR library and its commands. The AVR library is a set of Functions that are used in programming to interact with the hardware of the robot.

The Atmel studio is used to write and compile the program. The .hex file created is programmed into the robot using a AVR Programmer.

IDE required : Atmel Studio 6.1.

Operating system: windows 8.

C-library: AVR library.

2.9 Environment Required:

The robot is used to solve a Maze using its five reflectance sensors. So, the maze used in this purpose consists of the track that is made of a black lines which make the sensors identify the track by the amount of reflectance from the track. The track is placed on a white background so that the track is distinguishable from the base.

CHAPTER 3 - Maze Solving Robot

A Maze solving robot is a robot that is used to solve the given maze in an environment. The robot starts at any point on the maze and traverses through the track to reach the specific destination.

The robot in its first run , travels throughout the maze and learns the maze. It records the dead ends as unreachable paths and remembers the turns it has taken at the intersections.

The robot using its five reflectance sensors senses the track a black colored line and follows it until a dead end is encountered and notes it an unreachable path. The intersections are also recorded and the turns taken are remembered. Finally, first run is completed when the robot reaches the destination successfully.

The path stored remembered is simplified for avoiding unnecessary turns. In its second run , the robot travels through the track in the simplified path. The robot thus reaches the destination.

In a line maze contest, robots travel as quickly as possible along the lines from a designated start to the goal, keeping track of the intersections that they pass along the way. Robots are given several chances to run the maze, so that they can follow the fastest possible path after learning about all of the dead ends.

The mazes that the robot deals has a special feature: they have *no loops*. That is, there is no way to re-visit any point on the maze without retracing your steps.

The Maze Solving problem is solved using an algorithm **"Left Hand On The Wall"**.

3.1 Left Hand on the Wall Algorithm:

The basic strategy for solving a non-looped maze is called “left hand on the wall”. Imagine walking through a real labyrinth – a human-sized maze built with stone walls – while keeping your left hand on the wall at all times. You’ll turn left whenever possible and only turn right at an intersection if there is no other exit. Sometimes, when you reach a dead end, you’ll turn 180 degrees to the right and start walking back the way you came. Eventually, as long as there are no loops, your hand will travel along each length of wall in the entire labyrinth exactly once, and you’ll find your way back to the entrance. If there is a room somewhere in the labyrinth with a monster or some treasure, you’ll find that on the way, since you’ll travel down every hallway exactly twice.

CHAPTER 4 - Implementation

The Maze Solving robot problem is implemented using a program that is responsible for the robot to solve the maze .

The programming is done in C language using AVR library. The program is divided into multiple files according to their functionality.

The implementation starts with an initialization of the robot

4.1 Initialization: The robot is first initialized,

```
pololu_3pi_init(2000);
```

This must be called at the beginning of 3pi code, to set up the sensors. We use a value of 2000 for the timeout, which corresponds to 2000*0.4 us = 0.8 ms on our 20 MHz processor.

The important aspect of the initialization is that the sensors are calibrated for the first time and the calibrated values are stored in the array of five elements sensors[5];

```
for(counter=0;counter<80;counter++)
        {
                if(counter < 20 || counter >= 60)
                        set_motors(40,-40);
                else
```

```
            set_motors(-40,40);

        // This function records a set of sensor readings and keeps
        // track of the minimum and maximum values encountered.  The
        // IR_EMITTERS_ON argument means that the IR LEDs will be
        // turned on during the reading, which is usually what you
        // want.
        calibrate_line_sensors(IR_EMITTERS_ON);

        // Since our counter runs to 80, the total delay will be
        // 80*20 = 1600 ms.
        delay_ms(20);
    }
```

In the initialization phase , the robot uses characters for displaying the welcome note and the other information including sensors readings , battery reading, etc These characters are stored in the program space of the robot's memory. At the time of initialization the characters for displaying sensors reading on the lcd screen are loaded using

```
void load_custom_characters();
```

The calibrated sensor values are displayed on the lcd screen of the robot using

```
void display_readings(const unsigned int *calibrated_values)
```

function of the bargraph.c file.

After the robot is initialized using initialize() function , the main() function in “main.c” file calls the maze-solve() function for the maze to get solved.

```
// set up the 3pi
initialize();
// Call maze solving routine.
maze_solve();
```

The actual maze solving is done in the “maze-solve.c” file, which consists of the maze-solve() function.

4.2 Maze Solving:

The strategy of the program is expressed in the file “ maze-solve.c” . Most importantly, we want to keep track of the path that we have followed, so we define an array storing up to 100;
We also need to keep track of the current path length so that we know where to put the characters in the array.

```
char path[100] = "";
unsigned char path_length = 0; // the length of the path
```

The actual maze solving is done in maze-solving() function.

```
 void maze_solve()
{
    // Loop until the maze is solved.
    while(1)
    {
        // FIRST MAIN LOOP BODY
        follow_segment();

        // Drive straight a bit.  This helps us in case we entered the
        // intersection at an angle.
        // Note that we are slowing down - this prevents the robot
        // from tipping forward too much.
        set_motors(50,50);
        delay_ms(50);

        // These variables record whether the robot has seen a line to the
        // left, straight ahead, and right, whil examining the current
        // intersection.
        unsigned char found_left=0;
        unsigned char found_straight=0;
```

```
unsigned char found_right=0;

// Now read the sensors and check the intersection type.
unsigned int sensors[5];
read_line(sensors,IR_EMITTERS_ON);

// Check for left and right exits.
if(sensors[0] > 100)
        found_left = 1;
if(sensors[4] > 100)
        found_right = 1;

// Drive straight a bit more - this is enough to line up our
// wheels with the intersection.
set_motors(40,40);
delay_ms(200);

// Check for a straight exit.
read_line(sensors,IR_EMITTERS_ON);
if(sensors[1] > 200 || sensors[2] > 200 || sensors[3] > 200)
        found_straight = 1;
```

```
// Check for the ending spot.
// If all three middle sensors are on dark black, we have
// solved the maze.
if(sensors[1] > 600 && sensors[2] > 600 && sensors[3] > 600)
        break;

// Intersection identification is complete.
// If the maze has been solved, we can follow the existing
// path.  Otherwise, we need to learn the solution.
unsigned char dir = select_turn(found_left, found_straight, found_right);

// Make the turn indicated by the path.
turn(dir);

// Store the intersection in the path variable.
path[path_length] = dir;
path_length ++;

// You should check to make sure that the path_length does not
// exceed the bounds of the array.  We'll ignore that in this
// example.
```

```
    // Simplify the learned path.
    simplify_path();
  // Display the path on the LCD.
    display_path();
}
```

In this function , the follow_segment() function is called initially where the robot is made to follow the black track until an intersection or a dead is encountered . A dead end is detected using the following:

```
if(sensors[1] < 100 && sensors[2] < 100 && sensors[3] < 100)
  {
      // There is no line visible ahead, and we didn't see any
      // intersection.  Must be a dead end.
      return;
  }
```

If the sensor readings of the middle three sensors are less than 100 then a dead end is encountered and the call is returned to the maze-solve() function to take a ‘u-turn’.

An intersection is detected using:

```
else if(sensors[0] > 200 || sensors[4] > 200)
  {
      // Found an intersection.
```

```
return;
}
```

If the readings of the left most sensor is a greater value of 200 or more then there is a line on its left side, which is an intersection , also if reading on the right sensor is more than 200 then it is a intersection to its right.

When the call is returned to the maze-solve() function , the robot is slowed down a bit and the type of the intersection is detected based on its left and right sensor values.

If a left intersection is detected, then a variable found_left is set to 1.

```
if(sensors[0] > 100)
found_left = 1;
```

If a left intersection is detected, then a variable found_right is set to 1.

```
\
if(sensors[4] > 100)
found_right = 1;
```

And the robot is driven further more in low speed to check for any straight line.If a straight line is detected then a found_straight variable is set to 1.

```
if(sensors[1] > 200 || sensors[2] > 200 || sensors[3] > 200)
```

```
found_straight = 1;
```

Further, a destination reached condition is checked to stop the maze solving. A strong reading on the all five sensors is used to detect the destination. Then the maze solving is stopped and the path is stored in the path variable .

```
if(sensors[1] > 600 && sensors[2] > 600 && sensors[3] > 600)
    break;
```

Then all the above variables are passed toselect_ turn() function which returns a direction to be taken by the robot.

```
unsigned char dir = select_turn(found_left, found_straight, found_right);
```

The select_turn() function is used to direct the robot in the appropriate direction.

```
char select_turn(unsigned char found_left, unsigned char found_straight, unsigned char
found_right)
{
    // Make a decision about how to turn.  The following code
    // implements a left-hand-on-the-wall strategy, where we always
    // turn as far to the left as possible.
    if(found_left)
        return 'L';
```

```
        else if(found_straight)
                return 'S';
        else if(found_right)
                return 'R';
        else
                return 'B';
}
```

This dir variable is used to take the turn by using turn() function.

```
    void turn(char dir)
{
        switch(dir)
        {
        case 'L':
                // Turn left.
                set_motors(-80,80);
                delay_ms(200);
                break;
        case 'R':
                // Turn right.
                set_motors(80,-80);
```

```
            delay_ms(200);
            break;
        case 'B':
            // Turn around.
            set_motors(80,-80);
            delay_ms(400);
            break;
        case 'S':
            // Don't do anything!
            break;
        }
}
```

The turn function turns the robot in appropriate direction by using the set_motors() function where the parameters are the speeds of the left and right wheel respectively.

In order to take a left turn , a differential drive concept is used where the left wheel speed is set to -80 and the right wheel is set to 80 . Similarily to take right turn, 80,-80 is given.
To take a ‘u-turn’ the set_motors() function is executed for 400 ms where its completes a u-turn.

This loop is executed until the destination is reached and at every turn the robot records the direction or the turn taken in a array “path[100]” and the length of the path is increased for every turn.

```
path[path_length] = dir;
path_length ++;
```

This completes the first important infinite loop where the robot learns the maze.The maze is solved now and ready to traverse the maze in the definite path.

For the robot to traverse the maze in definite path it needs another infinite loop, the second important loop.

In this, the robot uses a simplified path of the path array.

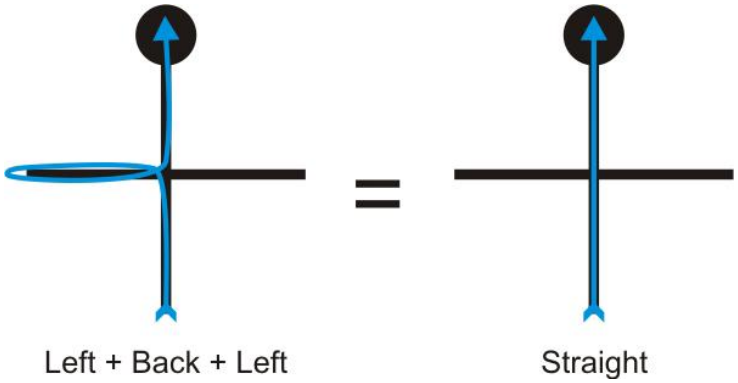
The path is simplified using the simplify_path() where all the dead ends and unwanted ends are simplified.

```
 void simplify_path()
{
    // only simplify the path if the second-to-last turn was a 'B'
    if(path_length < 3 || path[path_length-2] != 'B')
        return;

    int total_angle = 0;
    int i;
    for(i=1;i<=3;i++)
```

```
{
        switch(path[path_length-i])
        {
        case 'R':
                total_angle += 90;
                break;
        case 'L':
                total_angle += 270;
                break;
        case 'B':
                total_angle += 180;
                break;
        }
}

// Get the angle as a number between 0 and 360 degrees.
total_angle = total_angle % 360;

// Replace all of those turns with a single one.
switch(total_angle)
{
case 0:
```

```
                path[path_length - 3] = 'S';
                break;
            case 90:
                path[path_length - 3] = 'R';
                break;
            case 180:
                path[path_length - 3] = 'B';
                break;
            case 270:
                path[path_length - 3] = 'L';
                break;
            }

            // The path is now two steps shorter.
            path_length -= 2;
    }
```

If the path array has xBx as the occurring in its last 3 places , then the 3 places are replaced by the single direction depending on the total_angle variable.



Now the robot travels in the maze in a definite path after it is simplified. The robot just uses the follow_segment() function used before. Whenever an intersection is encountered the robot checks the path[path_length] array for relevant direction to be taken.

Finally , the robot ends up at the destination. Now the Robot can be run any times which solves the given maze.

The complete program that the robot requires is:

4.3 Program code

4.3.1 main.c:

```
#include <pololu/3pi.h>
#include <avr/pgmspace.h>
```

```
#include "bargraph.h"
#include "maze-solve.h"

// Introductory messages.  The "PROGMEM" identifier causes the data to
// go into program space.
const char welcome_line1[] PROGMEM = " Pololu";
const char welcome_line2[] PROGMEM = "3\xf7 Robot";
const char demo_name_line1[] PROGMEM = "Maze";
const char demo_name_line2[] PROGMEM = "solver";

const char welcome[] PROGMEM = ">g32>>c32";
const char go[] PROGMEM = "L16 cdegreg4";

// Initializes the 3pi, displays a welcome message, calibrates, and
// plays the initial music.
void initialize()
{
        unsigned int counter;
        unsigned int sensors[5];

        pololu_3pi_init(2000);
        load_custom_characters(); //
```

```
print_from_program_space(welcome_line1);
lcd_goto_xy(0,1);
print_from_program_space(welcome_line2);
play_from_program_space(welcome);
delay_ms(1000);

clear();
print_from_program_space(demo_name_line1);
lcd_goto_xy(0,1);
print_from_program_space(demo_name_line2);
delay_ms(1000);

// Display battery voltage and wait for button press
while(!button_is_pressed(BUTTON_B))
{
        int bat = read_battery_millivolts();

        clear();
        print_long(bat);
        print("mV");
        lcd_goto_xy(0,1);
```

```
        print("Press B");


        delay_ms(100);
}

wait_for_button_release(BUTTON_B);
delay_ms(1000);

// Auto-calibration: turn right and left while calibrating the
// sensors.
for(counter=0;counter<80;counter++)
{
        if(counter < 20 || counter >= 60)
                set_motors(40,-40);
        else
                set_motors(-40,40);

        calibrate_line_sensors(IR_EMITTERS_ON);
   // Since our counter runs to 80, the total delay will be
        // 80*20 = 1600 ms.
        delay_ms(20);
}
```

```
    set_motors(0,0);


    // Display calibrated values as a bar graph.
    while(!button_is_pressed(BUTTON_B))
    {
        // Read the sensor values and get the position measurement.
        unsigned int position = read_line(sensors,IR_EMITTERS_ON);


        clear();
        print_long(position);
        lcd_goto_xy(0,1);
        display_readings(sensors);
        delay_ms(100);
    }
    wait_for_button_release(BUTTON_B);
    clear();
    print("Go!");
    play_from_program_space(go);
    while(is_playing());
}
```

```
int main()
{
        initialize();
        maze_solve();

       while(1);
}
```

4.3.2 maze_solve.c:

```
#include <pololu/3pi.h>
#include "follow-segment.h"
#include "turn.h"

char path[100] = "";
unsigned char path_length = 0; // the length of the path

// Displays the current path on the LCD, using two rows if necessary.
void display_path()
{
```

```
        path[path_length] = 0;
        clear();
        print(path);

        if(path_length > 8)
        {
                lcd_goto_xy(0,1);
                print(path+8);
        }
}

// This function decides which way to turn during the learning phase of
// maze solving.
char select_turn(unsigned char found_left, unsigned char found_straight, unsigned char
found_right)
{
        // Make a decision about how to turn.  The following code
        // implements a left-hand-on-the-wall strategy, where we always
        // turn as far to the left as possible.
        if(found_left)
                return 'L';
        else if(found_straight)
```

```
            return 'S';
        else if(found_right)
            return 'R';
        else
            return 'B';
}

// Path simplification.  The strategy is that whenever we encounter a
// sequence xBx, we can simplify it by cutting out the dead end.

void simplify_path()
{
        // only simplify the path if the second-to-last turn was a 'B'
        if(path_length < 3 || path[path_length-2] != 'B')
            return;

        int total_angle = 0;
        int i;
        for(i=1;i<=3;i++)
        {
            switch(path[path_length-i])
            {
```

```
            case 'R':
                total_angle += 90;
                break;
            case 'L':
                total_angle += 270;
                break;
            case 'B':
                total_angle += 180;
                break;
            }
    }

    // Get the angle as a number between 0 and 360 degrees.
    total_angle = total_angle % 360;

    // Replace all of those turns with a single one.
    switch(total_angle)
    {
    case 0:
            path[path_length - 3] = 'S';
            break;
    case 90:
```

```
                path[path_length - 3] = 'R';
                break;
            case 180:
                path[path_length - 3] = 'B';
                break;
            case 270:
                path[path_length - 3] = 'L';
                break;
            }
          path_length -= 2;
}

void maze_solve()
{
      // Loop until we have solved the maze.
          while(1)
          {
              // FIRST MAIN LOOP BODY
                follow_segment();
               // Drive straight a bit.  This helps us in case we entered the
                // intersection at an angle.
                set_motors(50,50);
```

```
delay_ms(50);

// These variables record whether the robot has seen a line to the
// left, straight ahead, and right, whil examining the current
// intersection.
unsigned char found_left=0;
unsigned char found_straight=0;
unsigned char found_right=0;

// Now read the sensors and check the intersection type.
unsigned int sensors[5];
read_line(sensors,IR_EMITTERS_ON);

// Check for left and right exits.
if(sensors[0] > 100)
        found_left = 1;
if(sensors[4] > 100)
        found_right = 1;

// Drive straight a bit more - this is enough to line up our
// wheels with the intersection.
set_motors(40,40);
```

```
    delay_ms(200);

    // Check for a straight exit.
    read_line(sensors,IR_EMITTERS_ON);
    if(sensors[1] > 200 || sensors[2] > 200 || sensors[3] > 200)
        found_straight = 1;

    // Check for the ending spot.
    // If all three middle sensors are on dark black, we have
    // solved the maze.
    if(sensors[1] > 600 && sensors[2] > 600 && sensors[3] > 600)
        break;

    // Intersection identification is complete.
    // If the maze has been solved, we can follow the existing
    // path.  Otherwise, we need to learn the solution.
    unsigned char dir = select_turn(found_left, found_straight, found_right);

    // Make the turn indicated by the path.
    turn(dir);
  // Store the intersection in the path variable.
    path[path_length] = dir;
```

```
        path_length ++;

        // Simplify the learned path.
        simplify_path();
        // Display the path on the LCD.
        display_path();
}

// Solved the maze!

// Now enter an infinite loop - we can re-run the maze as many
// times as we want to.
while(1)
{
        // Beep to show that we finished the maze.
        set_motors(0,0);
        play(">>a32");

        // Wait for the user to press a button, while displaying
        // the solution.
        while(!button_is_pressed(BUTTON_B))
        {
```

```
            if(get_ms() % 2000 < 1000)
            {
                clear();
                print("Solved!");
                lcd_goto_xy(0,1);
                print("Press B");
            }
            else
                display_path();
            delay_ms(30);
        }
        while(button_is_pressed(BUTTON_B));
        delay_ms(1000);
        int i;
        for(i=0;i<path_length;i++)
        {
            // SECOND MAIN LOOP BODY
            follow_segment();

            // Drive straight while slowing down, as before.
            set_motors(50,50);
            delay_ms(50);
```

```
                set_motors(40,40);
                delay_ms(200);

                // Make a turn according to the instruction stored in
                // path[i].
                turn(path[i]);
            }

            follow_segment();

        }
}
```

4.3.3 follow_segment.c:

```
  #include <pololu/3pi.h>
  void follow_segment()
{
        int last_proportional = 0;
        long integral=0;

        while(1)
```

```
{
    // Get the position of the line.
    unsigned int sensors[5];
    unsigned int position = read_line(sensors,IR_EMITTERS_ON);
    int proportional = ((int)position) - 2000;
    int derivative = proportional - last_proportional;
    integral += proportional;
    last_proportional = proportional;
    int power_difference = proportional/20 + integral/10000 + derivative*3/2;
    const int max = 60;

  if(power_difference > max)
        power_difference = max;
    if(power_difference < -max)
        power_difference = -max;

    if(power_difference < 0)
        set_motors(max+power_difference,max);
    else
        set_motors(max,max-power_difference);

    if(sensors[1] < 100 && sensors[2] < 100 && sensors[3] < 100)
```

```
            {
                // There is no line visible ahead, and we didn't see any
                // intersection.  Must be a dead end.
                return;
            }
            else if(sensors[0] > 200 || sensors[4] > 200)
            {
                // Found an intersection.
                return;
            }
    }
}
```

4.3.4 bar_graph.c:

```
#include <pololu/3pi.h>
#include <avr/pgmspace.h>

// Data for generating the characters used in load_custom_characters
// and display_readings.  By reading levels[] starting at various
// offsets, we can generate all of the 7 extra characters needed for a
// bargraph.  This is also stored in program space.
```

```
const char levels[] PROGMEM = {
        0b00000,
        0b00000,
        0b00000,
        0b00000,
        0b00000,
        0b00000,
        0b00000,
        0b11111,
        0b11111,
        0b11111,
        0b11111,
        0b11111,
        0b11111,
        0b11111
};

// This function loads custom characters into the LCD.  Up to 8
void load_custom_characters()
{
        lcd_load_custom_character(levels+0,0);
        lcd_load_custom_character(levels+1,1);
```

```
        lcd_load_custom_character(levels+2,2);
        lcd_load_custom_character(levels+3,3);
        lcd_load_custom_character(levels+4,4);
        lcd_load_custom_character(levels+5,5);
        lcd_load_custom_character(levels+6,6);
        clear();}

void display_readings(const unsigned int *calibrated_values)
{
        unsigned char i;

        for(i=0;i<5;i++) {

                const char display_characters[10] = {' ',0,0,1,2,3,4,5,6,255};
.
                char c = display_characters[calibrated_values[i]/101];

                // Display the bar graph character.
                print_character(c);
        }
}
```

4.3.5 turn.c:

```
#include <pololu/3pi.h>
void turn(char dir)
{
        switch(dir)
        {
        case 'L':
                // Turn left.
                set_motors(-80,80);
                delay_ms(200);
                break;
        case 'R':
                // Turn right.
                set_motors(80,-80)
       delay_ms(200);
                break;
        case 'B':
                // Turn around.
                set_motors(80,-80);
                delay_ms(400);
                break;
```

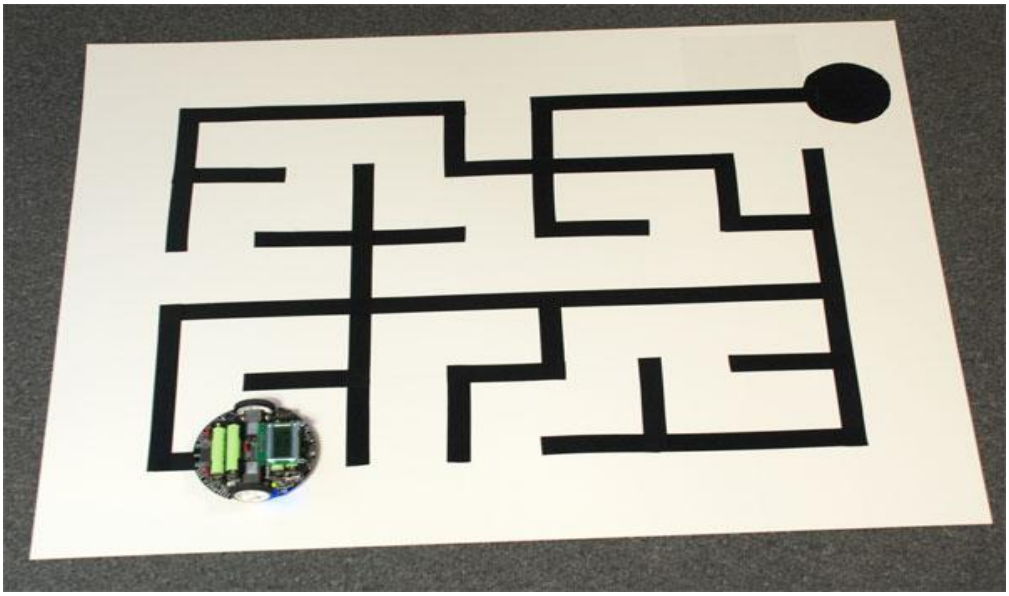
```
        case 'S':
            // Don't do anything!
            break;
        }
}
```

CHAPTER 5 – Experimental Results

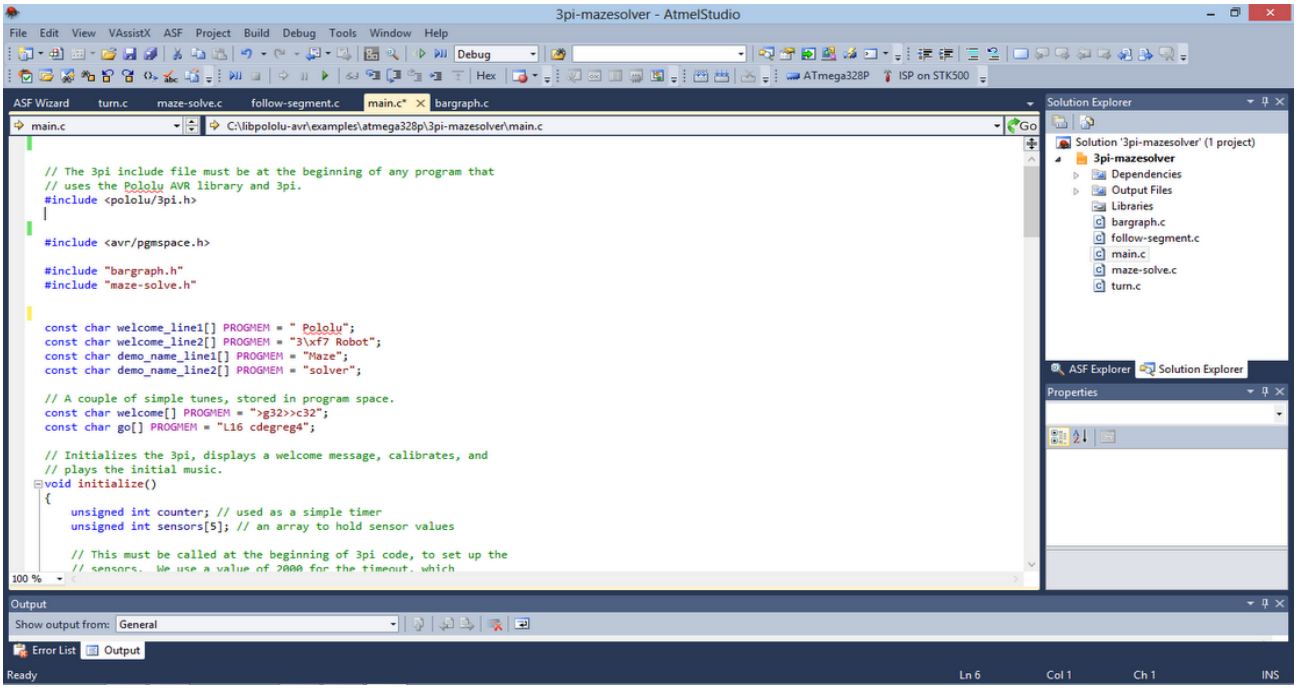
5.1 Screenshots

5.1.1. The assembled pololu's 3pi robot with batteries and the a welcome note

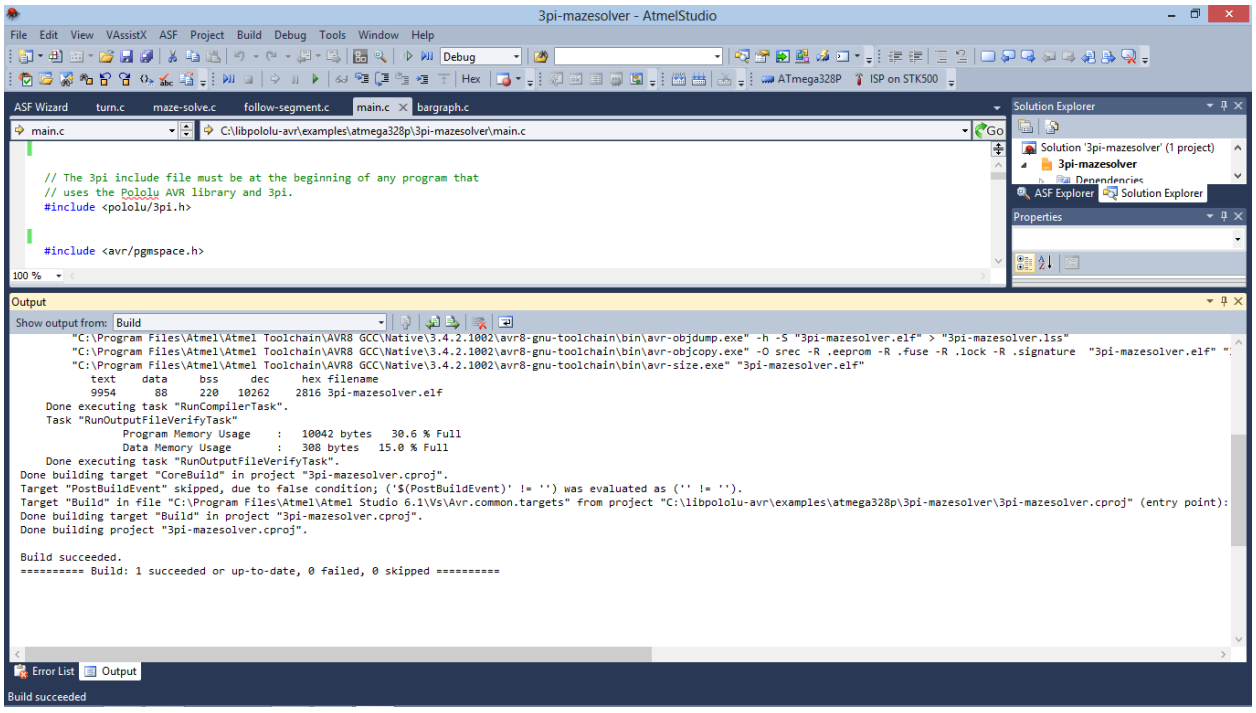
5.1.2. The Maze environment which the robot is required to solve. The robot is at its initial position.



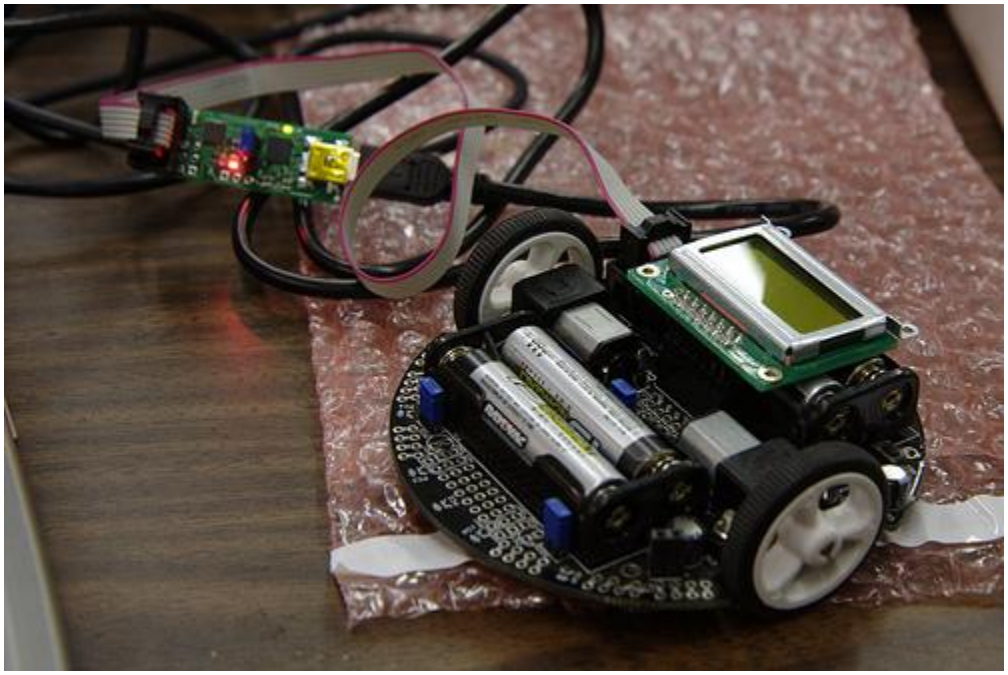
5.1.3. The Atmel Studio 6.1 showing solution explorer and the different c files of the program.



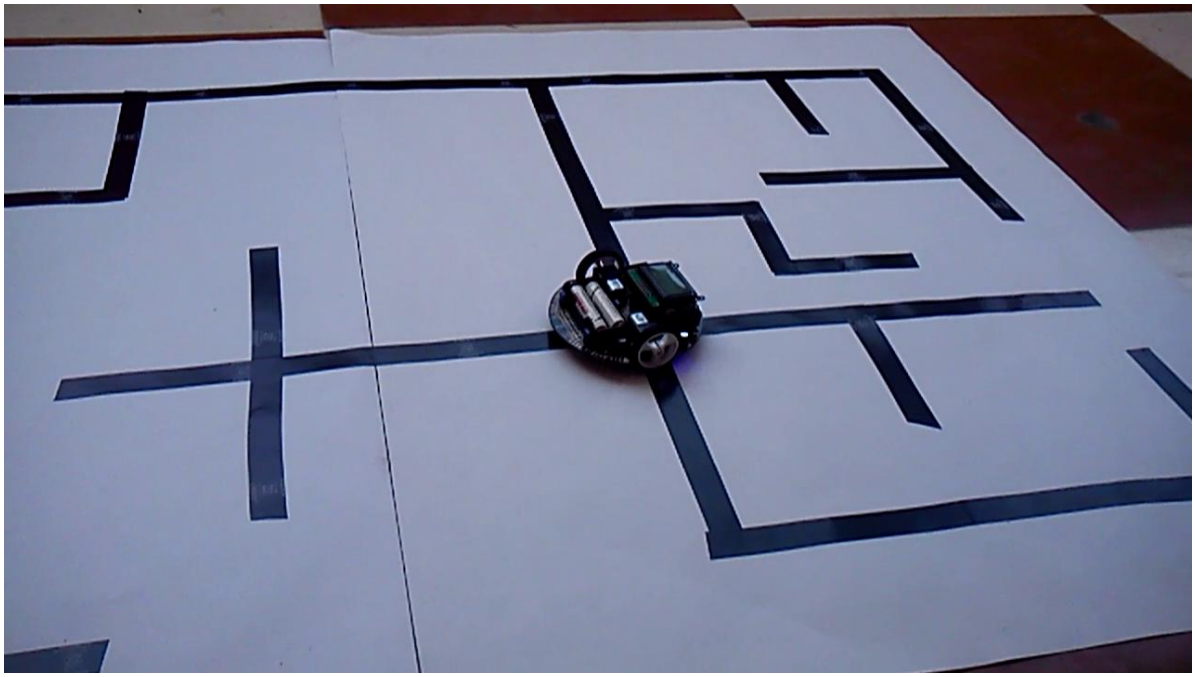
5.1.4.The Atmel studio Showing successful build of the 3pi maze-solving project.



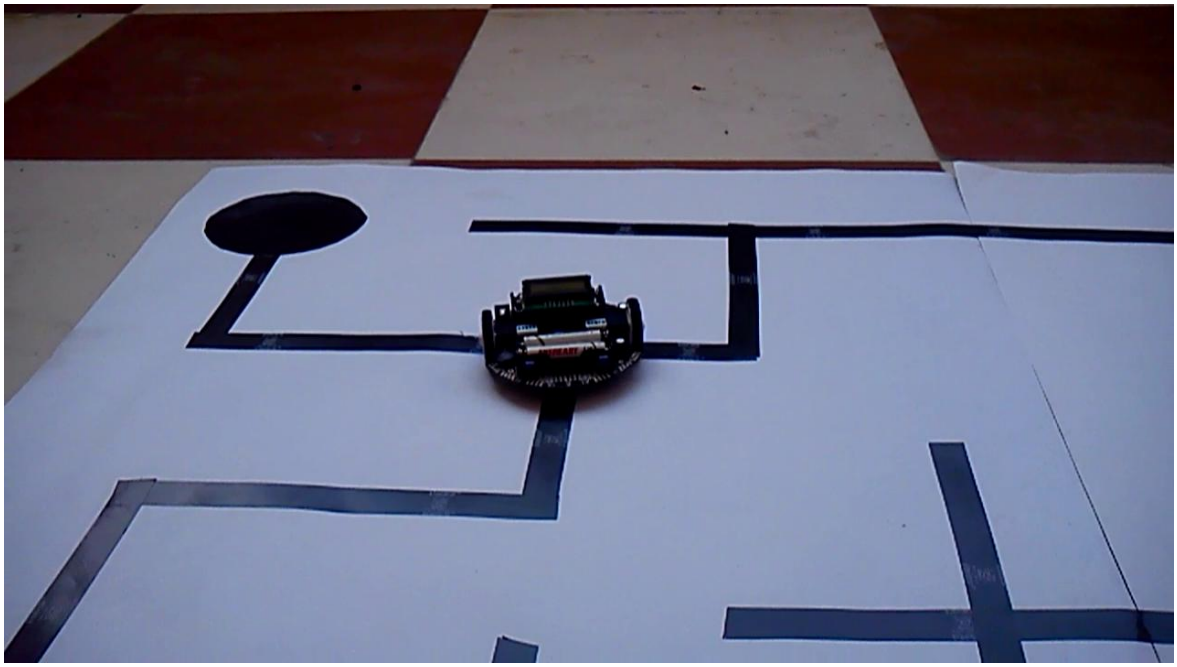
5.1.5.The Robot is being programmed by the AVR USB Programmer . It burns the .hex file generated by Atmel Studio after building the project.



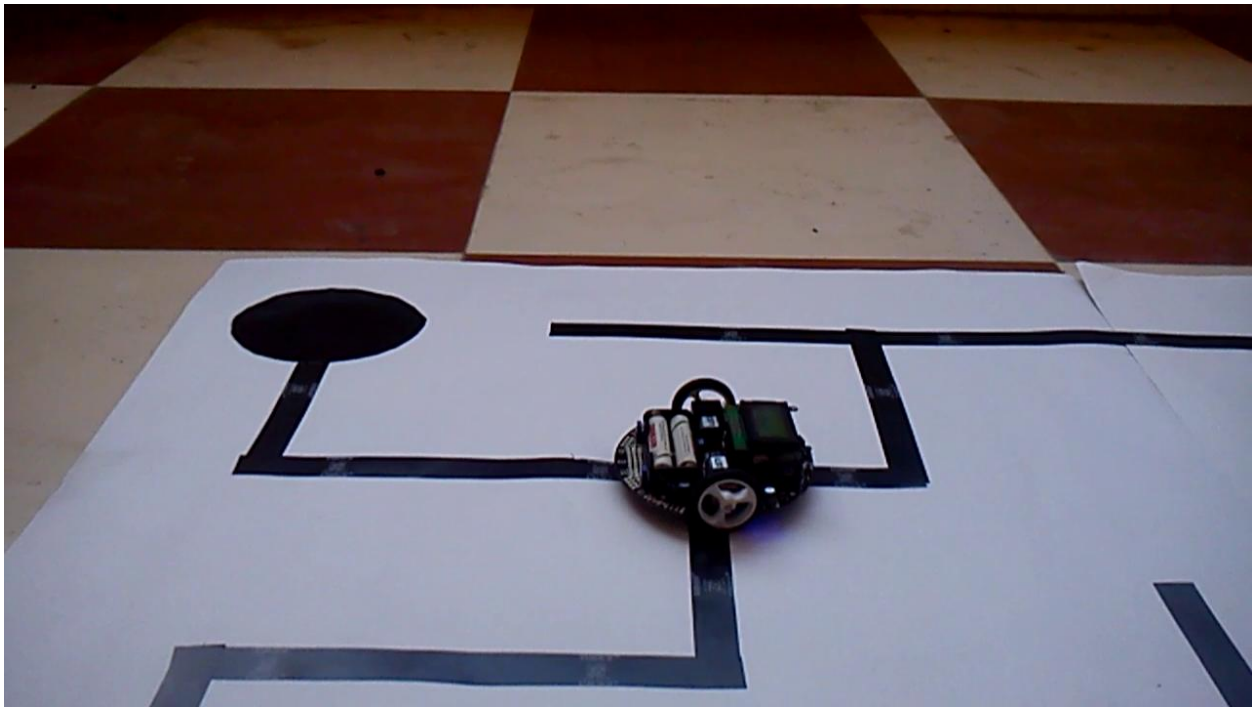
5.1.6.The 3 pi robot solving a maze. The robot takes a left at the intersection by using left hand on the wall in its first run.



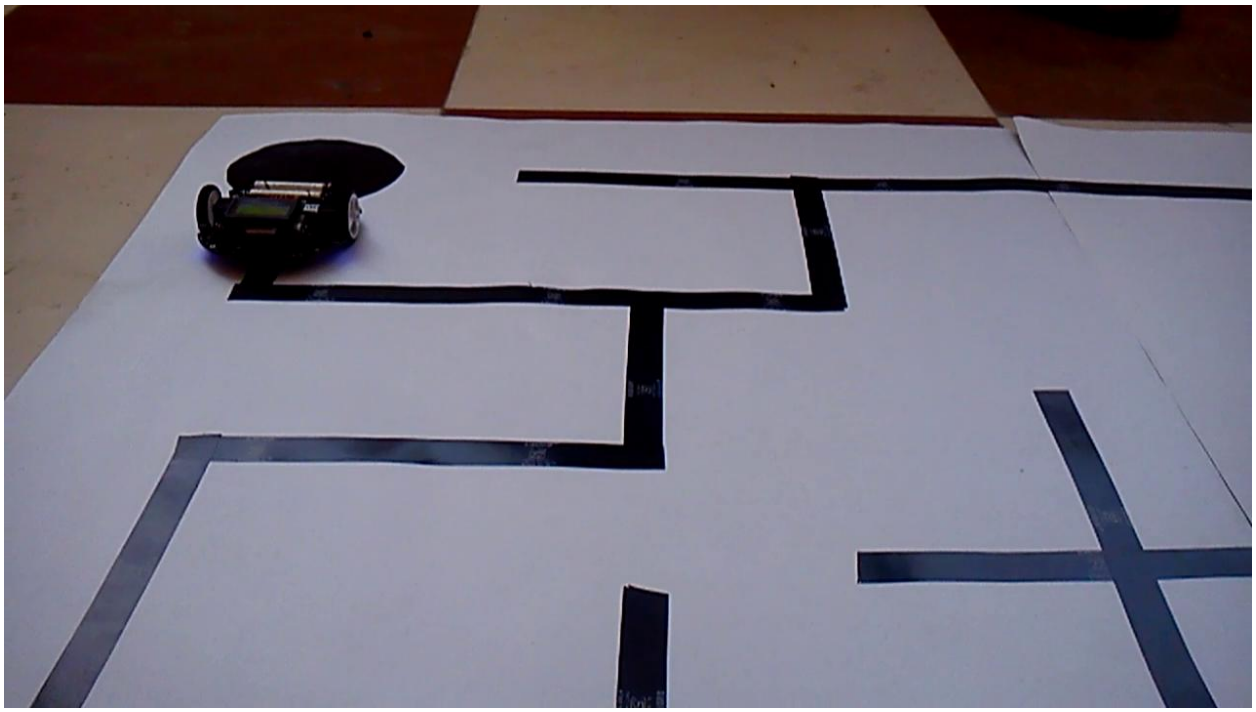
5.1.7. The robot taking another left at an intersection in its first run.



5.1.8.The robot goes in a straight path heading to the destination in its second run.



5.1.9.robot finally reaches the destination where all the sensors read the blackline in highamount.



CHAPTER 6 - CONCLUSION AND FUTURE WORK

6.1 Conclusion:

This project deals with the successful maze solving using a 3pi robot. The robot uses the left hand on the wall algorithm to solve the maze and uses the sensors to identify the path. The threshold values of the sensors are taken differently at different instances of the maze solving phase of the robot. These different threshold values are used for certain implications.

This maze solving robot can be used in situations where the identification of the feasible path from the source to the destination is difficult by general perception. The Maze is solved in the first run and the robot traverses the maze in a definite and optimal path in its subsequent runs.

6.2 Future work:

In this project, we address the problem of solving a maze using the left hand on the wall algorithm .The functionality of the robot can be further increased by implementing much better and advanced algorithms. In this system the robot uses sensors in order to identify the path and the destination points, other functionalities such as video cameras can also be installed in order to identify the path by visual perception. The robot with the help of sensors could identify the end points in the path, but with some modifications in the code it can be also made to identify obstacles in its path along with the end points. Thus it can be made to work in a dynamic environment with obstacles. The scope of the dynamic environment is large depending on the nature of the obstacles, i.e static or dynamic in nature. Thus the maze solving robot can be used to solve the maze in much more complex and dynamic situations in a more intelligent manner.

These robots with obstacle detection using Distance Measuring Sensors can be used for developing equipment that can be used by blind people . With some sophisticated algorithms and more advanced sensors , the robot can be used in path detection and following in Real World where they can be used to supply ammo in military bases or to mine the coal in Coal Mines.

CHAPTER 7 – REFERENCES

7.1 List of references

1. Abelson; diSessa (1980), *Turtle Geometry: the computer as a medium for exploring mathematics*
2. Public conference, December 2, 2010 - by professor Jean Pelletier-Thibert in Academie de Macon (Burgundy - France) - (Abstract published in the Annals academic, March 2011 - ISSN: 0980-6032)

   Charles Tremaux (° 1859 - † 1882) Ecole Polytechnique of Paris (X:1876), French engineer of the telegraph
3. Édouard Lucas: *Récréations Mathématiques* Volume I, 1882.
4. H. Fleischner: *Eulerian Graphs and related Topics.* In: *Annals of Discrete Mathematics* No. 50 Part 1 Volume 2, 1991, page X20.
5. Even, Shimon (2011), *Graph Algorithms* (2nd ed.), Cambridge University Press, pp. 46–48, ISBN 978-0-521-73653-4.
6. Sedgewick, Robert (2002), *Algorithms in C++: Graph Algorithms* (3rd ed.), Pearson Education, ISBN 978-0-201-36118-6.