RidgeSoft IntelliBrain-Bot
Small-Wheeled Robot

Creighton Evans
Gregory Bontrager
CEN 3212 Embedded Systems Programming
Dr. Janusz Zalewski
Florida Gulf Coast University
April 25, 2007
# Table of contents

Section 1: Introduction 3  
Section 2: Problem Specification 4  
Section 3: Design Specification 6  
Section 4: Implementation and Testing 9  
Section 5: Conclusion 24  
Section 6: List of References 25
Section 1: Introduction

Since the time of Archytas of Tarentum in Ancient Greece [1], man has been mesmerized by robotic automatons. From simple mechanical birds to Honda’s ASIMO [2], mankind finds itself experimenting with robots throughout the ages in attempts at making life easier and more productive. Robotics is the scientific study of the design, manufacture, and application of robots. An engineer must have a working knowledge of electronics, mechanics, and software in order to properly create a working robot. Although the actual appearance and capabilities of robots vary widely, they all share the features of a mechanical, mobile structure that operates under some type of control. Robotics stretches the boundaries on the intelligence of binary, mechanical minds as well as the human minds of their creators. Throughout history, robotics has been an interest for many thinkers and has continuously evolved.

The basis for all modern day robotics has been to improve human life. Whether the work is too dirty, tedious, or precise for humans, developers continue creating improvements and evolving the science around modern robotics [3]. Looking at the history of robotics, the work of science fiction authors like Isaac Asimov have inspired developers in the idea of creating robots with the same intelligence of their human creators [4]. This idea was not new, but this inspiration has led robotics down the road of creating artificial intelligence. Massachusetts Institute of Technology, MIT, has opened departments designed for the continuing improvement of robotics and artificial intelligence [4]. Though most robots are simplistic in nature, the Mars Rovers [5] are living laboratories, taking samples from Mars, and analyzing them before sending the data back to Earth all the while sending back pictures of the Mars surface.

The purpose of this project is to become acquainted with robots and the programming involved in creating a working automaton. This project will be using the Ridgesoft IntelliBrain-Bot which is a small-wheeled robot using the java programming language. It comes with its own IDE called RoboJDE along with the Software and Documentation needed to run the robot. The CPU for the robot is 16-bit based Atmel ATmega128 with 132K RAM. It also has a 16x2 liquid crystal display for easily displaying actions and other messages programmed for the robot unit. It is a simple unit,
however, without any past experience, using the IntelliBrain-Bot will offer a new robotics experience for emerging developers.

Section 2: Problem Specification

The Ridgesoft IntelliBrain-Bot includes the parts in the picture below taken from the Ridgsoft website.

The most crucial elements to the robot are of course the metal chassis, the controller, its sensors, display module, and of course the nuts, bolts, and wheels that keep it together. A diagram of the robot given for this project can be seen below.
Listed in light green is the connection to the servos. The cables are connected through a hole in the middle of the chassis to the servos controlling the wheels. The connections are made in ports S1, left, and S2, right servos. Connecting them properly on the controller ensures that the wheels should function correctly. The same thing applies to the left and right infrared photo reflector sensors. The connection to the controller, in a light blue, and the sensors themselves, in a darker blue, are necessary for our project since they will be used to detect an obstacle in the robot's path. The sensors are screwed into the side of the metal chassis, next to the wheels. This way they can properly detect an obstruction. The connections are made in ports A4, left, and A5, right sensors. The LCD display module, in a light magenta, is easily connected in the four-pin slot on the controller. Now messages can be displayed while the robot is in action. The nine-pin serial port in orange is perhaps one of the most important parts of the robot where the program from the RoboIDE is sent to the robot for testing. For the project, a cable needed for the transfer was also provided. Not listed in the diagram is the box in the top left of the diagram where you can see a red and black cable going connecting. That box connects the power supply which is on the bottom of the chassis. The robot takes four AA batteries which is easily obtainable for both testing and the end project. All these essential connectors are part of the IntelliBrain-Bot controller given in the Brain Bot Package. It is the meat of the robot and controls the flow of data. All of these parts together make the IntelliBrain-Bot.
For this project, the problem is what to program this little robot to do. For the solution, it was decided to look at the Roomba Robotic Vacuum Cleaner. The Roomba is programmed to go over an entire room while avoiding obstacles in its path. The IntelliBrain-Bot will take on a similar program. For a given distance, the robot will move in a straight forward motion. If an obstacle obstructs its path, the program will interrupt and execute a function to move the robot backwards a given short distance, make a ninety degree turn to the right, move forward a given short distance, and then make another ninety-degree turn to the right. Basically, for the interrupt, if an obstruction is found, the robot will turn around and go the other way. Once the robot travels on its path for a given distance, a function will be called to make a ninety degree turn and repeat the primary function. Hence the robot will repeat the same path in a perpendicular direction to the original. Only two iterations will be allowed for the single turn, allowing the robot to travel through an entire room avoiding all obstructions at least once.

Section 3: Design Specification

In Section 2, a simple summary of the program was explained. In this section, more analysis will be given. Our robot will follow the same fundamental program as the Roomba Vacuum Cleaner. Stage one of the program will have the robot navigate a room until it reaches a wall. Stage 2 moves the robot in the same way as in Stage 1 but in a perpendicular direction. Together they cover the entire area of the room.

![Stage 1 Diagram](image1)

![Stage 2 Diagram](image2)

Stage 1  
Stage 2
Following some discussion, it was agreed that the key to proper function would be collision detection. We decided that the best way to accomplish this would be to derive the value of a Boolean variable from regular samples of feedback from the robot’s wheel motion sensors. The sensors work by sending out a beam that bounces back if it strikes a spoke but otherwise passes through. As the wheels turn rapidly, the readings should alternate between “high” (no spoke) and “low” (spoke). Thus, if the reading remains the same for an extended period of time, it can be deduced that a collision has occurred since such an event causes the wheels to stop moving.

After some preliminary brainstorming, a flowchart was developed to aid in the construction of the robot program. After checking Boolean values and the use of counters, the robot detects a wall by traversing its length twice before beginning its perpendicular motion.
The initial coding could not be easily implemented into any other IDEs besides the RoboJDE that came with the IntelliBrain-bot. However, some preliminary coding can be given here.

```java
public boolean collision;

public class CollisionDetection {
    public void run() {
        while (collision is FALSE) {
            check wheels, if wheels are stopped turn to true
        }
    }

    void Forward () {
        move robot forward
    }

    void Turn () {
        turn 90 degrees
    }

    void Backward () {
        move robot back
    }

    void Reverse () {
        Backward();
        for (Conditions met) {
            Turn();
            Forward();
        }
    }
}
```
void Primary () {
    while (collision is TRUE) {
        Reverse();
    }
    Forward();
}

void main () {
    new Coverage();
    Primary();
    new CollisionDetection();
}

The next step for now is to develop our pseudo-code and begin testing our robot.

Section 4: Implementation and Testing

As a group, the online tutorials proved useful in understanding how the IntelliBrain-Bot is supposed to work. Unfortunately, the first attempts using the robot met with failure as the RobotJDE would not compile acceptable code. Not even simple code to move the robot forward would compile. It was discovered that example code was included with the RobotJDE program in its Program Directory. Using the example code, the code still failed to compile. Upon careful study on the ridgesoft website, there were two apparent changes to the IntelliBrain-Bot that was discovered. First, the RobotJDE was upgraded from version 1.3 to 1.5. Even after downloading the newer version, there were still some complications with the code. The code now compiled, but some of the code in the examples did not work with the IntelliBrain-Bot. The second discovery was the replacement of the IntelliBrain-Bot Controller. It was upgraded to a better version and is on sale on their website, however without the proper time, it was decided to modify our code to the version our controller understood. Thankfully the RobotJDE version 1.5 proved useful to us as it was now suited with Java 1.4.2.
Having resolved these issues, we moved on to writing simple programs of our own and testing them. Our attempt to make the robot move forward was met with success. More coding yielded a program that would make the robot move in a path that formed a square. This pivotal success told us that our ultimate goal was now within our reach. Our most daunting challenge was designing an algorithm for collision detection. The sensors given only detected the movement of the wheels. It was thought if the wheels collided on a surface, they would stop if met with an obstacle. Then we would only have to detect if the same signal appeared numerous times and turn the robot around. This proved to be a problem since the servos themselves were quite strong and, despite a collision, would continue rotating. The next option would be hard coding the distance, but since the object of the experiment is to cover any room entirely, that made no sense for us to include that. Hard-coding the distance would hinder the robot’s versatility and convenience to a hypothetical user, this option was dismissed. Since the robot did not possess any other sensors to use, it was decided to do the best we could with what we had.

The first step was to get the robot partially coded with the operation. We used simple classes to control the simple movements so they did not need to be repeated and only writing the class would be sufficient.

```java
static void MoveForward()
{
    leftServo.setPosition(60);
    rightServo.setPosition(40);
}

static void MoveBackward()
{
    leftServo.setPosition(40);
    rightServo.setPosition(60);
}

static void TurnLeft()
{
    leftServo.setPosition(0);
```
rightServo.setPosition(0);
}

static void TurnRight()
{
    leftServo.setPosition(100);
    rightServo.setPosition(100);
}

static void AllStop()
{
    leftServo.off();
    rightServo.off();
}

The reason MoveForward and MoveBackward contain 60 and 40 in their positions is that upon study it was found that power was given to the servos primarily between 36 and 64. Any thing above or below did not influence power to the servos. It was thought that maybe we could use this to control the wheels and detect the collision more easily. As such we continued with the code. We created our main class to begin running the mainPath class.

    static void mainPath()
    {
        if(counter!=2)
        {
            try
            {
            int[] results = new int[480];
            // drive forward
            MoveForward();
            int i = 0;
            int rightLimit=475;
            int leftLimit=40;
            while(i < 480)
            {
                int leftSample=leftWheelInput.sample();
            
            
        }}
int rightSample=rightWheelInput.sample();
if(leftSample < 475 && rightSample < 40)
{
    AllStop();
    HitObstacle();
    break;
}
Thread.sleep(125);
i++;
}
}
catch(InterruptedException e){}
}

This is not the finalized code of the class since it is still being used to test the operation. However, it can be seen that we have a while loop to control the time it takes the robot to move forward. Given that there is no interruption on the robot for a given minute, it will halt operation. It was decided to implement this if the robot were to be used in an open field or a very large room. It also checks the counter. The counter cannot be two or else it would have crossed the room three times in that period. Twice is all that is needed. Counter is also a global variable along with the Analog Inputs for the wheels as well as the Servos since they are used throughout the application. If there is a collision as detected by the sensors, the program calls the class HitObstacle.

    static void HitObstacle()
    {
        try
        {
            MoveBackward();
            Thread.sleep(500);
            AllStop();

            if(turn==true)
                // rotate clockwise approximately 90 degrees
                TurnRight();
            else
                TurnLeft();
        }
        catch(InterruptedException e){}
    }
// rotate counter-clockwise approximately 90 degrees

    TurnLeft();

    Thread.sleep(625);
    AllStop();
    MoveForward();

    int i = 0;
    int rightLimit=rightWheelInput.sample();
    int leftLimit=leftWheelInput.sample();
    while(i < 5)
    {
        int leftSample=leftWheelInput.sample();
        int rightSample=rightWheelInput.sample();
        if(leftSample<leftLimit&&rightSample<rightLimit)
        {
            MoveBackward();
            Thread.sleep(200);
            AllStop();

            if(turn==true)
                // rotate clockwise approximately 90 degrees
                TurnRight();
            else
                // rotate counter-clockwise approximately 90 degrees
                TurnLeft();

            Thread.sleep(625);
            counter++;
            break;
        }
        else
        {
            Thread.sleep(50);
            i++;
        }
    }
Simply put, \textit{HitObstacle} will back the robot up a bit and proceed with two turns in the same motion. It will change a Boolean variable after those turns so that the robot will know in which direction to turn. If a collision is made in the process of turning the robot and moving forward a bit, the robot will have detected it has reached a wall and will proceed in making a third turn as well as incrementing the counter.

In our final testing stage, both robots that were tested had faulty hardware. The one given to Greg Bontrager actually had the sensors misplaced on the robot and would not read the data correctly. Unfortunately without the appropriate tools, the robot given to Creighton Evans was used to implement the final code.
import com.ridgesoft.io.*;
import com.ridgesoft.robotics.*;
import com.ridgesoft.intellibrain.*;
import java.lang.*;
import java.util.*;
import java.io.*;

public class Robot {

    static Servo leftServo = IntelliBrain.getServo(1);
    static Servo rightServo = IntelliBrain.getServo(2);
    static int counter = 0;
    static boolean turn = true;
    static AnalogInput leftWheelInput = IntelliBrain.getAnalogInput(4);
    static AnalogInput rightWheelInput = IntelliBrain.getAnalogInput(5);

    static void MoveForward() {
        //MoveForward is used to Position the servos in moving forward
        static void MoveForward()
{
    leftServo.setPosition(100);
    rightServo.setPosition(0);
}

//MoveBackward is used to Position the servos in moving backward
static void MoveBackward()
{
    leftServo.setPosition(0);
    rightServo.setPosition(100);
}

//TurnLeft is used to Position the servos in turning left
static void TurnLeft()
{
    leftServo.setPosition(0);
    rightServo.setPosition(0);
}

//TurnRight is used to Position the servos in turning right
static void TurnRight()
{
    leftServo.setPosition(100);
    rightServo.setPosition(100);
}

//AllStop is used to turn the servos off
static void AllStop()
{
    leftServo.off();
}
rightServo.off();

///////////////////////////////////////////////////////////////////////////////////////////////////////
// mainPath                                             //
// This method is used to begin moving the bot the same way a             //
// Roomba vacuum cleaner moves. It detects if the bot meets an          //
// obstacle and calls the method HitObstacle. Otherwise, the          //
// method ends if the bot continues moving over a minute,               //
// ending the program. This is to prevent the bot from being            //
// set in open areas and moving off into the distance.                 //
///////////////////////////////////////////////////////////////////////////////////////////////////////
static void mainPath()
{
      
      // Check if the global counter is equal to 2. The              //
      // counter is used to check if the robot has hit a wall          //
      // (not an obstacle). This allows the bot to move in             //
      // a perpendicular direction, however we do not want             //
      // the same movements occurring. So we limit counter to          //
      // two.                                                          //
      //                                                                  
      if(counter!=2)
      {
            
            //Try-Catch for Sleep Interruption
            try
            {

                     // Make the bot begin moving forward
                     MoveForward();
                     Thread.sleep(1000);
// Initialize variables for testing collision in while loop
int i = 0;

//right/leftLimit are used to take first sample data
int rightLimit=rightWheelInput.sample();
int leftLimit=leftWheelInput.sample();

// While loop iterates 398 times for a total of one minute
while(i < 398)
{
    //Sleep for the wheel to change signals
    Thread.sleep(150);

    //Take new sample data
    int rightSample = rightWheelInput.sample();
    int leftSample = leftWheelInput.sample();

    // This if statement checks if both sample data are the same. If they are, then the robot's wheels are stopped upon collision and needs to avoid the obstacle.
    if (leftSample <= leftLimit+20 &&
        leftSample >= leftLimit-20 &&
        rightSample <= rightLimit+20 &&
        rightSample >= rightLimit-20)
    {
        System.out.println("Obstacle Found");
    }
AllStop();
HitObstacle();
break;
}

// When the if statement is false, the Limits are
// updated so the next sample
// data can be checked.
rightLimit=rightSample;
leftLimit=rightSample;
i++;

}
}
catch(InterruptedException e){}
}

////////////////////////////////////////////////////////////////////////////////////////////////////
// HitObstacle        
// This method is used to avoid an obstacle in the robot's path.    
// Initially the bot makes two turns in the same direction as    
// determined by the global boolean, turn.  Once the second   
// turn is made, the robot changes the boolean to its opposite.   
// However, if a wall is found when the bot moves foward,        
// the counter is incremented, and the bot makes a third turn.   
// When the method ends, mainPath is called to continue running./
////////////////////////////////////////////////////////////////////////////////////////////////////
static void HitObstacle()
{


try
{
    // Move robot backward for half a second
    // Since the robot did not encounter an obstacle while
    // moving forward, there should not be any obstacles
    // when backing up
    MoveBackward();
    Thread.sleep(500);
    AllStop();

    // Make the first initial turn
    if(turn==true)
        // rotate clockwise approximately 90 degrees
        TurnRight();
    else
        // rotate counter-clockwise approximately 90 degrees
        TurnLeft();

    // To make a 90 degree turn, Ridgesoft says to make
    // the thread sleep for 625 milliseconds.
    Thread.sleep(625);
    AllStop();

    // Move the robot forward for just over a second while
    // checking if it hits a wall
    MoveForward();
    Thread.sleep(600);

    // Similar to mainPath, the variables are initialized
    int i = 0;
int rightLimit=rightWheelInput.sample();
int leftLimit=leftWheelInput.sample();

// This time the while loop only makes 4 checks
while(i < 4)
{
    // Again, the sleep cycle needed to rotate the wheel to
    // the opposite position
    Thread.sleep(150);

    // Take new sample data
    int leftSample=leftWheelInput.sample();
    int rightSample=rightWheelInput.sample();

    // This if statement checks if both sample
    // data are the same. If they are, then the
    // robot's wheels are stopped upon collision
    // with a wall. The robot then backs up for
    // a shorter time period and makes an
    // additional turn. This puts the bot in a
    // perpendicular direction to its original path.
    // The counter is also incremented.
    if (leftSample <= leftLimit+20 &&
        leftSample >= leftLimit-20 &&
        rightSample <= rightLimit+20 &&
        rightSample >= rightLimit-20)
    {
        System.out.println("Wall Found");
    }
// Move the robot a short distance back
MoveBackward();
Thread.sleep(200);
AllStop();

// Make the additional turn
if(turn==true)
    // rotate clockwise approximately 90 degrees
    TurnRight();
else
    // rotate counter-clockwise 90 degrees
    TurnLeft();

// Increment the counter,
// Sleep for the appropriate time for
// a 90 degree turn
counter++;  
Thread.sleep(625);
break;
}

// When the if statement is false, the Limits are
// updated so the next sample
// data can be checked.
rightLimit=rightSample;
leftLimit=rightSample;
i++;
}
// Make the final turn
if (turn==true)
    // rotate clockwise approximately 90 degrees
    TurnRight();
else
    // rotate counter-clockwise approximately 90 degrees
    TurnLeft();

// Sleep for 90 degree turn and set boolean variable to its opposite
Thread.sleep(625);
turn=!turn;
mainPath();
}
catch(InterruptedException e){}
}

public static void main(String args[])
{
    // Initiate Roomba Movement
    System.out.println("Operation Begins");

    // Call mainPath to begin movement
    mainPath();

    // Halt any movement left over from functions
    AllStop();
    System.out.println("Operation End!");
}
} // End Robot
This operation takes sample code from the tutorials on Ridgesoft’s website and manipulates it to simulate what happens in a collision. Further coding and testing resulted in the discovery that, however sound our code may be, faulty hardware prevents the robot from functioning as originally intended. For one thing, the wheels never came to a full stop as mentioned before. The main impediment is that the treads on the wheels do not generate enough traction to stop the wheels’ rotation when a collision occurs. We had counted on the wheels ceasing to rotate when the robot struck an obstacle, and therefore our collision detection algorithm was based on detecting whether or not the wheels were moving. Even after replacing the treads on Creighton’s robot, the speed of the servos prevented the wheels from coming to a complete halt.

It was attempted to reduce power to the servos so that the wheels would stop. As documented in the “Programming Your Robot to Navigate” tutorial, a position of 50 would result in a neutral position or no movement. The robot did not comply with this since it still showed the minimal amount of movement therefore the servos were obviously malfunctioning. When the position was adjusted for the slowest possible settings, 51 and 49, the right wheel tended to be slower than the left one. Attempting to compensate made the right wheel faster while the left was now slower. Slowing the robot did not seem to work. Still, while testing the code and the robot on full speed, the code worked so long as the sensors were covered or the turns were slowed, but it still met with failure as obviously the speed of the servos had been affected. Where the robot should maneuver a perfect ninety degree turn, the servos would either be too fast or slow and adjusting them proved similar issues when it was attempted to compensate for speed.

It was at this time that we allowed our code to be examined by those who have worked on the IntelliBrain-Bot before, both online and off, and they agreed that the robot had faulty hardware issues. In a perfect world, the code would work soundly as the robot would come to a halt. Thus the repeating signals would result in a turn. Even with their honest opinions, the group continued trying to circumvent the issues with the hardware. Still the robot continues to experience hardware issues, and the program, though complete, cannot operate efficiently. This is perhaps analogous to what happens when a user attempts to run a programmer’s code on an incompatible system.
Section 5: Conclusion

The roomba operation, though programming was a success, came to a failure due to a faulty system. It is unfortunate that the robot could not be completed, and for later projects, it could be possible to order replacement parts and perhaps upgrade the IntelliBrain-Bot to its latest controller. Theoretically, embedded systems programmers should be prepared to face challenges presented by hardware architecture and find the best way to circumvent any initial problems it may present to the software they are designing. Even after having our suspicions about the hardware confirmed by other programmers, we made significant efforts to do exactly that, but through the experiment, the group came to realize that, despite the greatest programming efforts, no one could avoid faults in a system entirely. In the world of embedded systems, it is very important to be able to avoid faults entirely. For future projects with the IntellBrain-Bot, it may be wise to assure that the hardware is up-to-date, properly installed, and operational. It is recommended that the servos be replaced on Creighton’s robot since the servos were not working efficiently. The infrared sensors on Greg’s robot need to be moved to a new position, and his robot would need to be tested for further problems. Once the revisions to the robots have been made, it is highly recommended that the controller for both robots be upgraded to the newest version released by Ridgesoft. The memory has been expanded on the new board, and with better support for Java and new APIs, it would offer a better programming experience. Of course, upgrading and replacing faulty equipment, though theoretically sound to fix faults in a system, could potentially create more errors, but it would still be worth the effort rather than letting the robots remain as they are currently. For more ideas in the future, it would be interesting to purchase additional sensors on Ridgesoft’s website. In particular, there is an ultrasonic attachment that might be interesting to use in collision detection, two additional infrared sensors that detect obstacles in front of the robot, and a camera attachment. With these new sensors, the robot could potentially navigate an entire room filled with obstacles in three entirely different ways: sensor readings, sonic readings, and visual readings. Despite being
unable to complete the project completely, the experience with the IntelliBrain-Bot was especially useful for understanding the process in programming and testing a simple moving mechanical robot as opposed to simple compiling and running code on a PC.

**Section 6: List of References**